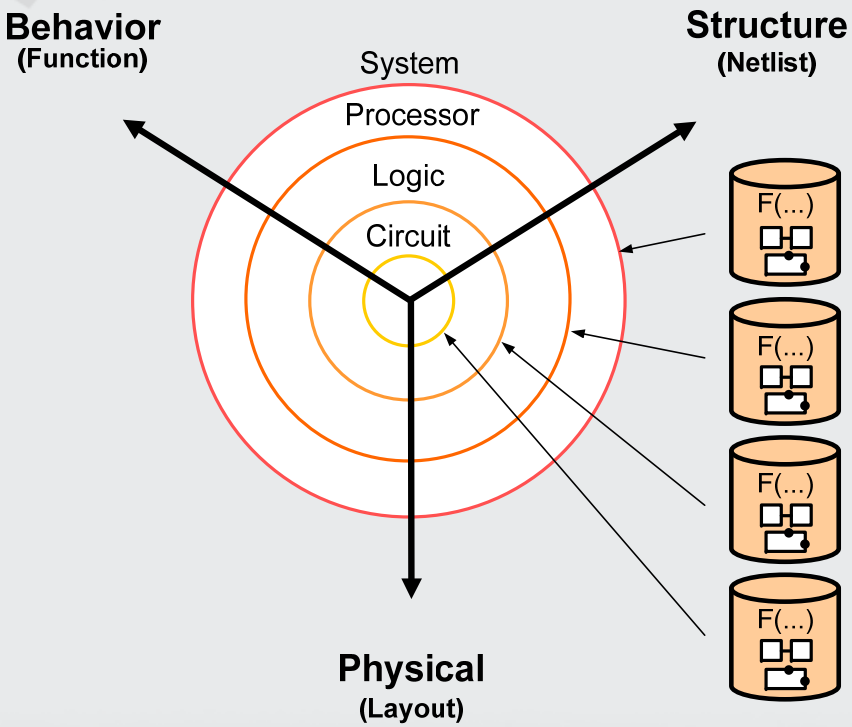
Lecture 5: Design Methodology & IP Modeling

(adapted with permission from UCI's Embedded Systems Design Course)

Design Methodology

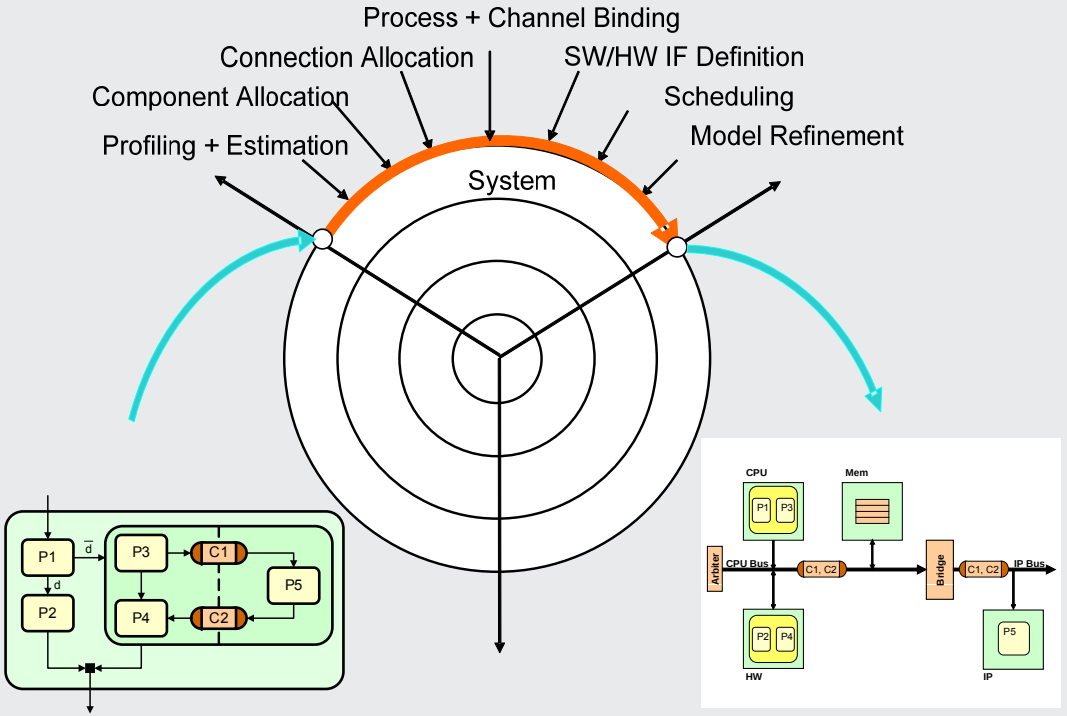
- Aims to design IP (models, components) or tools
- Depends on
  - ✓ application domain,
  - ✓ company, and
  - ✓ design group focus
- Evolves with
  - ✓ technology,
  - ✓ complexity, and
  - ✓ automation
- Standardization if treating special cases is expensive

Y Chart



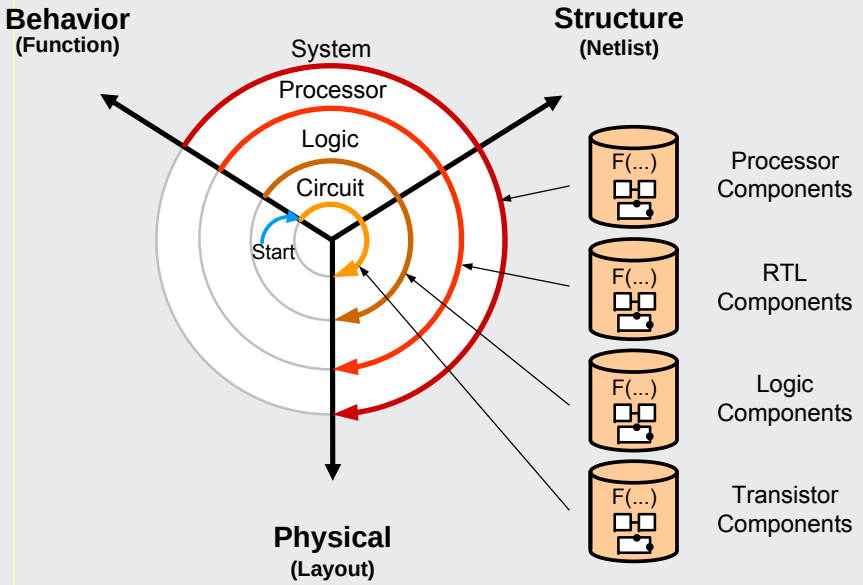
- 3 design views
  - ✓ Behavior, Structure, Physical
- 4 abstraction levels
  - ✓ Circuit, Logic, Processor, System
- 5 component libraries
  - ✓ Transistors
  - ✓ Logic (standard cells)
  - ✓ RTL (ALUs, RFs, …)
  - ✓ Processor (standard, custom)
  - ✓ System (multi-core with NoC)

System Synthesis



- Several tasks: different sequences possible
  - ✓ Different MoCs, different libraries, different features, different platforms
  - ✓ Different tools, different metrics, different quality

Bottom-Up Design



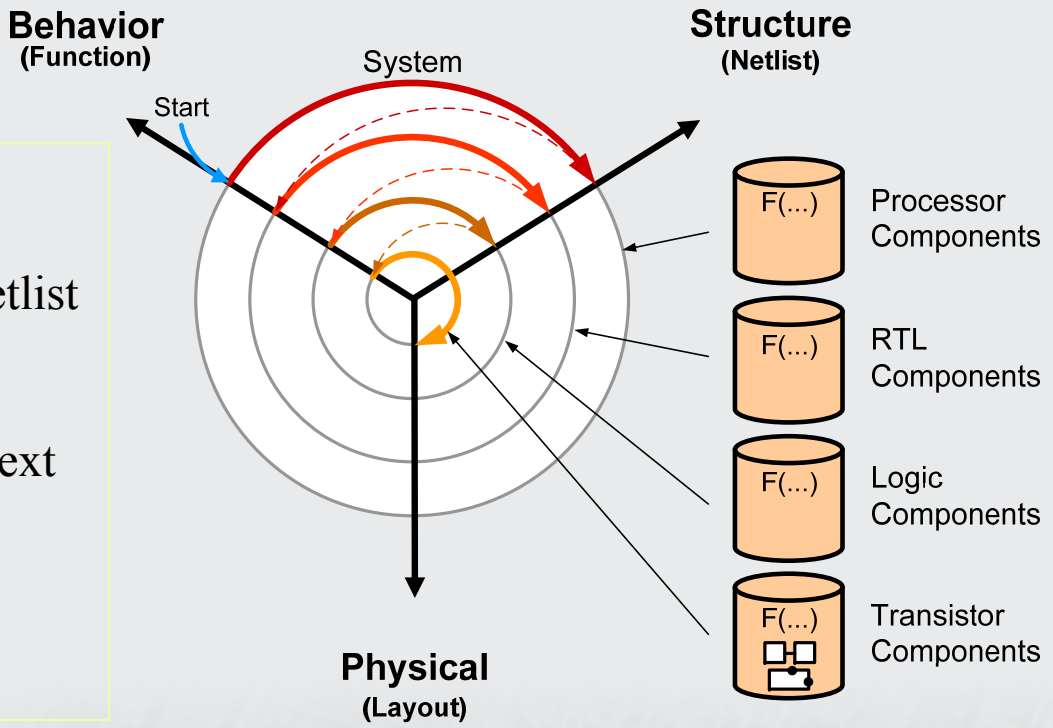
- Starts from the bottom level
- Each level generates a library for the next higher level
  - ✓ Circuit: logic cells
  - ✓ Logic: RTL components
  - ✓ Processor: system-level processing and communication
  - ✓ System: system platform

Bottom-Up Design

- Pros
  - ✓ Abstraction levels clearly separated
  - ✓ Accurate metric estimation at each level
  - ✓ Globally distributed development possible
  - ✓ Easy management
- Cons
  - ✓ An optimal library for each design is difficult to predict
    - o All possible components with all possible parameters
    - o All possible optimizations for all possible metrics
  - ✓ Library customization is outside the design group
  - ✓ Layout is performed at every level

Top-Down Design

- Starts with the top level
- Functional description is converted into component netlist at each level
- Each component function is decomposed further on the next abstraction level
- Layout is given only for transistor components

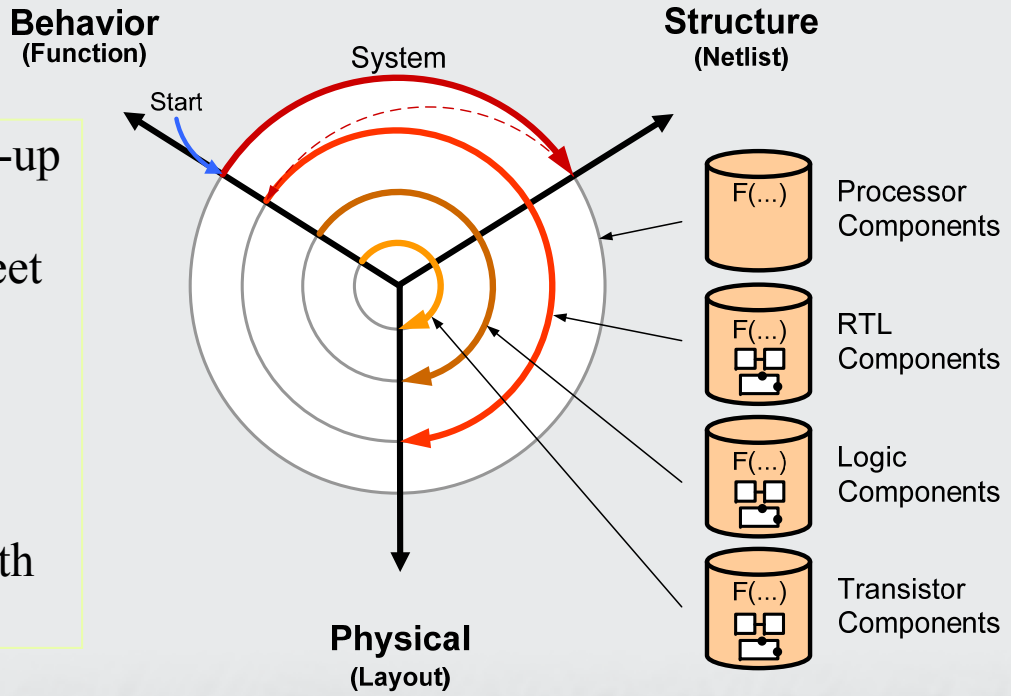


Top-Down Design

- Pros
  - ✓ Highest level of customization possible at each abstraction level
  - ✓ Only one small transistor library needed
  - ✓ Only one layout design at the end
- Cons
  - ✓ Difficult metric estimation at upper levels since layout not known
  - ✓ Design decision impact on higher level not clear
  - ✓ Hot spot removal is difficult
  - ✓ Metric annotation (closure) from lower to higher levels needed

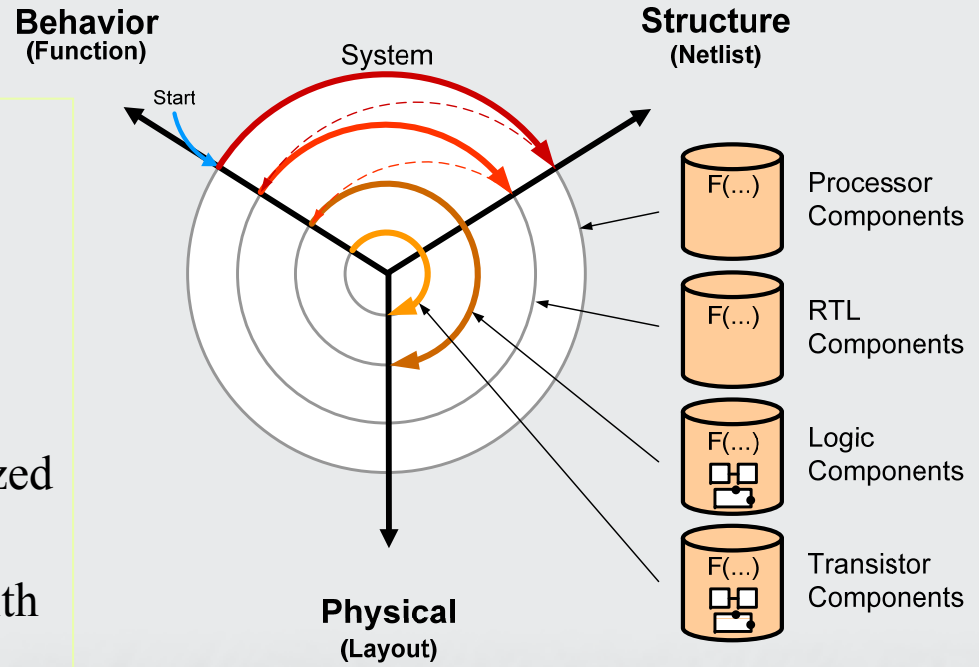
Meet in the Middle Design I

- Combines top-down &bottom-up
  - ✓ synthesis vs. layout compromise
- Processor level where they meet
- MoC is synthesized into processor components
- Processor components are synthesized with RTL library
- System layout is generated with RTL components



Meet in the Middle Design II

- RTL where they meet
- MoC synthesized with processor components
- Processor components are synthesized with RTL library
- RTL components are synthesized with standard cells
- System layout is performed with standard cells
- Two levels of layout

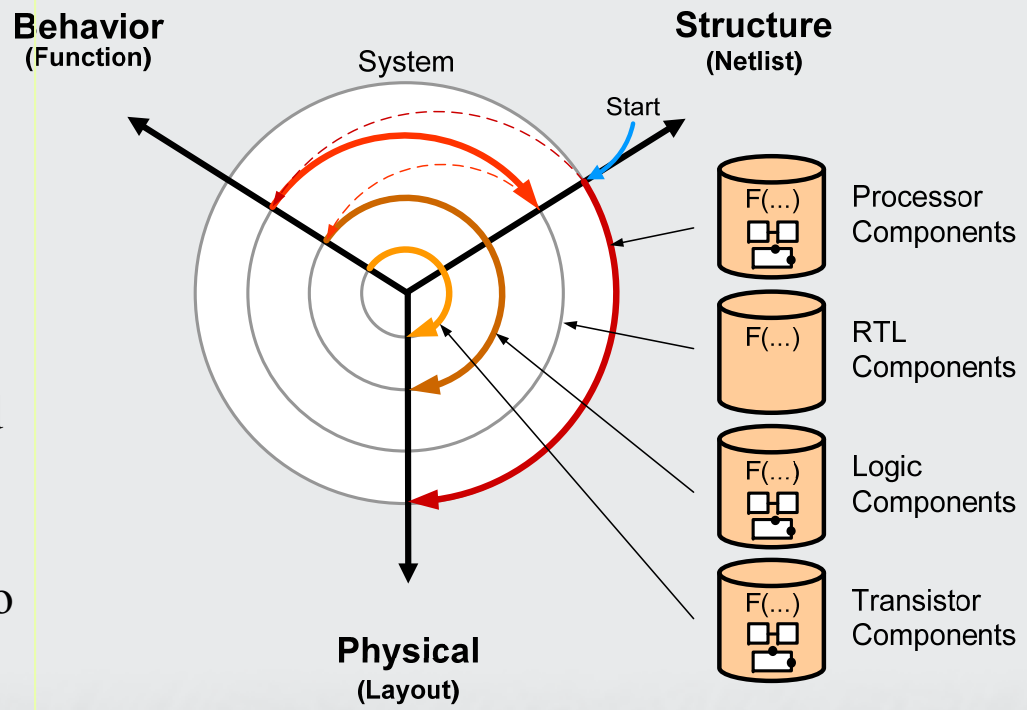


Meet in the Middle Design

- Pros
  - ✓ Shorter synthesis
  - ✓ Less layout
  - ✓ Less libraries
  - ✓ Better metric closure
- Cons
  - ✓ Still needs libraries
  - ✓ More then one layout
  - ✓ Metric closure still needed
  - ✓ Library components may not be optimal

Platform Methodology

- System platform with standard components and synthesizable custom components for application optimization
- Layout is on system level or predefined with special area for custom components layout
- Custom components synthesized with RTL and logic and laid out with standard cells
- Custom components must fit into platform structure

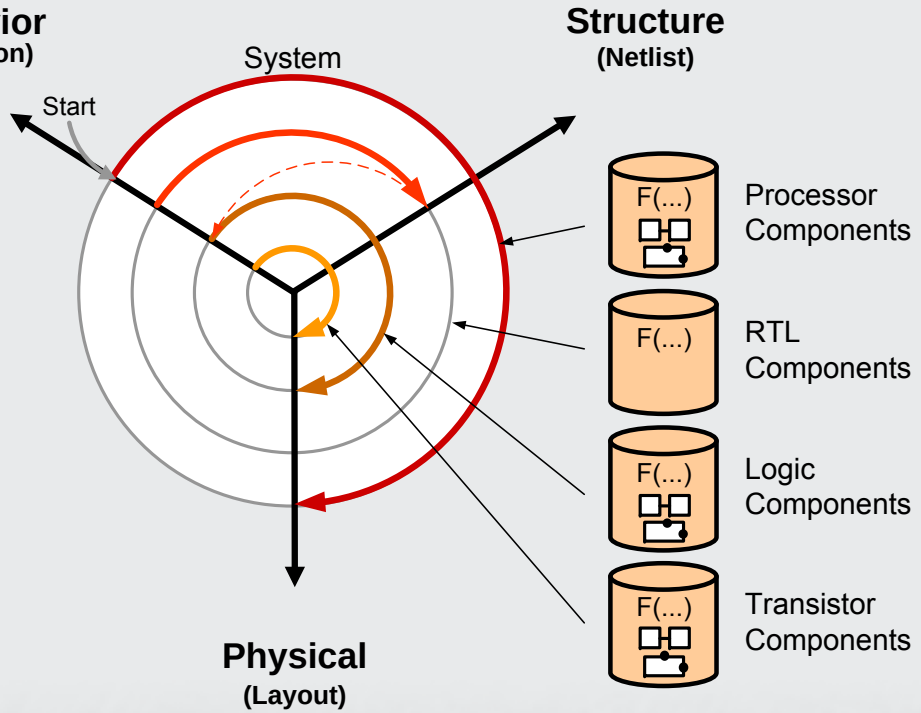


Platform Methodology

- Pros
  - ✓ Two types of layout: system layout for platform (could be predefined) and standard cell layout for custom components
  - ✓ Standard processors are available
  - ✓ Custom and interface components are added for optimization
- Cons
  - ✓ Platform customization is still needed
  - ✓ SW and IF components synthesis required

System Design

- Methodology for embedded systems developers
- System platform with architecture cells
- Layout on system level with architecture cells
- Architecture cells defined for specific application and design metrics
- Architecture cells pre-synthesized with RTL and logic and laid out with standard cells
- A retargetable compiler for architecture cells

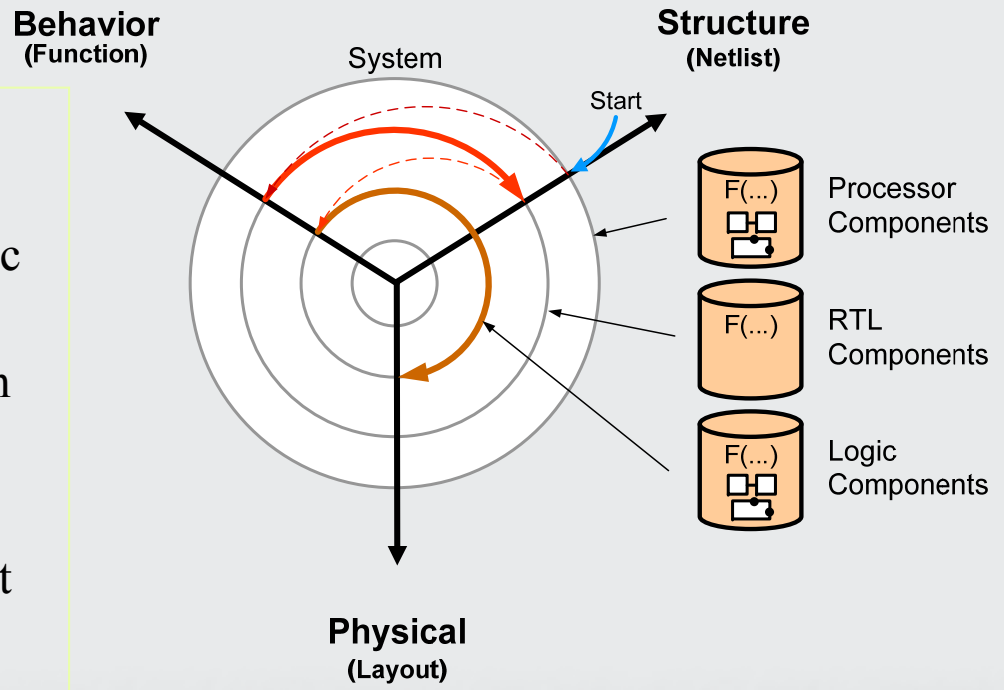


System Design

- Pros
  - ✓ Processor-level component only
  - ✓ Single retargetable compiler for all architecture cells
  - ✓ Processor-level layout
  - ✓ Methodology for application experts
  - ✓ Minimal knowledge of system and processor levels
- Cons
  - ✓ Architecture cell definition and library
  - ✓ IS definition
  - ✓ Change of mind

FPGA Design

- Starts with system structure
- Processor components synthesized with RTL and logic components
- Components implemented with LUT and BRAMs
- Layout only once
- Metric estimation very difficult
- Estimation is hidden in the FPGA supplier tools



Modeling

- Abstract view of a design
  - ✓ Representation of reality in each design step
    - o Apply analysis, synthesis and verification techniques
- Core of automated design flow
  - ✓ Varying levels of abstraction
    - o Level & organization of detail
  - ✓ Well-defined and unambiguous semantics
    - o Objects, composition rules and transformations
- Models of behavior (Models of Computation)
  - ✓ Concurrent computation
  - ✓ Communication
- Models of structure
  - ✓ Processing, storage and communication elements (PEs and Ces
  - ✓ NoC or bus

Models of Computation (MoCs)

- Conceptual, abstract description of system behavior
  - ✓ Classification based on underlying characteristics
    - o Computation and communication
  - ✓ Decomposition into pieces and their relationship
    - o Objects and composition rules
  - ✓ Well-defined, formal definition and semantics
    - o Functionality (data) and order (time)
      - ➢ Formal analysis and reasoning
      - ➢ Various degrees of complexity and expressiveness

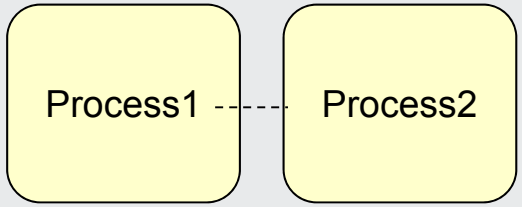
➢ Analyzability & expressiveness of behavioral models

Programming Models

- Imperative programming models
  - ✓ Ordered sequence of statements that manipulate program state
  - ➢ Sequential programming languages [C, C++, …]
- Declarative programming models
  - ✓ Dataflow based on explicit dependencies (causality)
  - ➢ Functional or logical programming languages [Haskell or Prolog]
- Synchronous programming models
  - ✓ Reactive vs. transformative: explicit concurrency
  - ✓ Lock-step operation of concurrent statement blocks
  - ➢ Synchronous languages [Esterel (imperative), Lustre (declarative)]

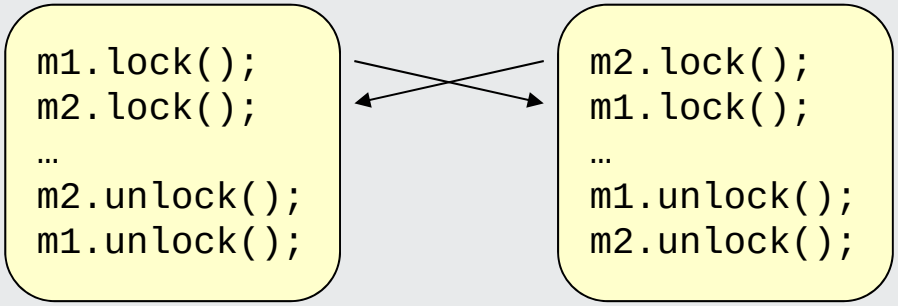
Process-Based Models

- Concurrency and causality (data flow)
- Set of processes
  - ✓ Processes execute in parallel
    - o Concurrent composition
  - ✓ Each process is internally sequential
    - o Imperative program
- Inter-process communication
  - ✓ Shared memory [ShMem]
    - o Synchronization: critical section/mutex, monitor, …
  - ✓ Message passing [MPI]
    - o Synchronous, rendezvous (blocking send)
    - o Asynchronous, queues (non-blocking send)
- Implementation: OS processes or threads
  - ✓ Single or multiple processors/cores



Deadlocks

- Circular chain of 2 or more processes which each hold a shared resource that the next one is waiting for
  - ✓ Circular dependency through shared resources



  - ✓ Prevent chain by using the same precedence
  - ✓ Use timeouts (and retry), but: livelock
- Dependency can be created when resources are shared
  - ✓ Side effects, e.g. when blocking on filled queues/buffers

Determinism

- Deterministic: same inputs always produce same results
- Random: probability of certain behavior
- Non-deterministic: undefined behavior (for some inputs)
  - ✓ Undefined execution order
    - o Statement evaluation in imperative languages: `f(a++, a++)`
    - o Concurrent process race conditions:

```
x = a;
y = b;
```

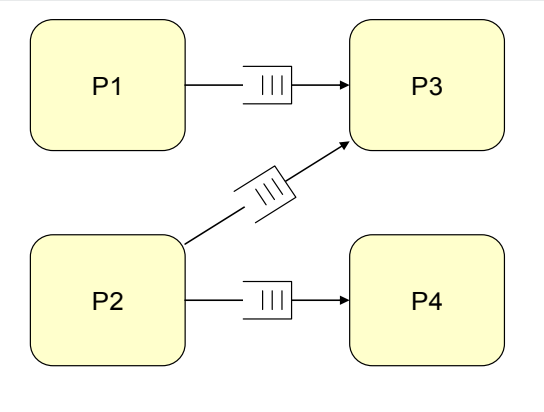
```
a = 1;
b = 2;
```

x = ?, y = ?

- ➢ Can be desired or undesired
  - ✓ How to ensure correctness?
    - o Simulator must typically pick one behavior
  - ✓ But: over-specification?
    - o Leave freedom of implementation choice

Kahn Process Network (KPN) [Kahn74]

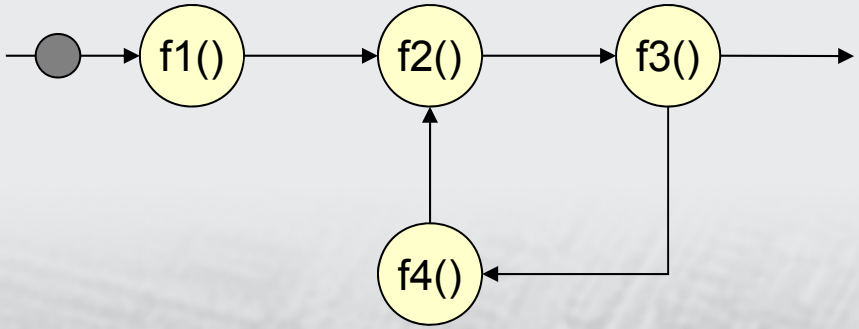
- C-like processes communicating via FIFO channels
  - ✓ Unbounded, uni-directional, point-to-point queues



- Deterministic
  - ✓ Behavior does not depend on scheduling strategy
  - ✓ Focus on causality, not order (implementation independent)
- Difficult to implement [Parks95]
  - ✓ Size of infinite FIFOs in limited physical memory?
  - ✓ Dynamic memory allocation, dependent on schedule
  - ✓ Boundedness vs. completeness vs. non-termination (deadlocks)

Dataflow

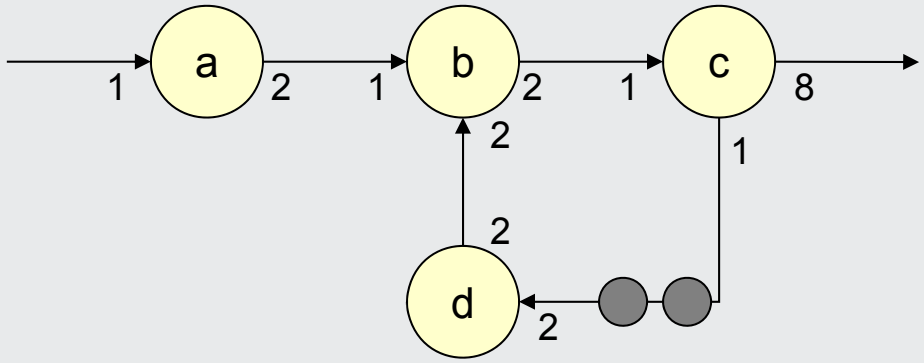
- Breaking processes down into network of *actors*
  - ✓ Atomic blocks of computation, executed when *firing*
  - ✓ Fire when required number of input *tokens* are available
    - o Consume required number of tokens on input(s)
    - o Produce number of tokens on output(s)
  - ✓ Separate computation & communication/synchronization
    - o Actors (indivisible units of computation) may fire simultaneously, any order
    - o Tokens (units of communication) can carry arbitrary pieces of data
- Unbounded FIFOs on arcs between actors



- Signal-processing applications

Synchronous Dataflow (SDF)

- Fixed number of tokens per firing
  - ✓ Consume fixed number of inputs
  - ✓ Produce fixed number of outputs



- Can be scheduled statically
  - ✓ Solve system of linear equations for relative rates
  - ✓ Periodically schedule actors in proportion to their rates
- Find a sequence of firings in each period
  - ✓ Trade-off code size and buffer sizes
    - o Single-appearance vs. memory-minimal schedule

Process Calculi

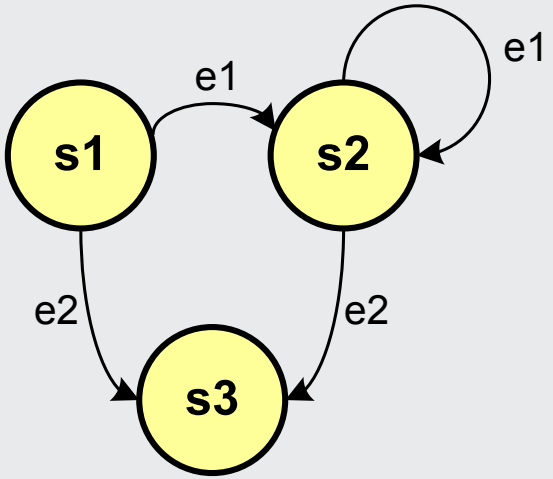
- Rendezvous-style, synchronous communication
  - ✓ Communicating Sequential Processes (CSP) [Hoare78]
  - ✓ Calculus of Communicating Systems (CCS) [Milner80]
  - ➢ Restricted interactions
- ➢ Formal, mathematical framework: process algebra
  - ✓ Algebra = <objects, operations, axioms>
    - o Objects: processes $\{P, Q, \ldots\}$, channels $\{a, b, \ldots\}$
    - o Composition operators: parallel ($P \| Q$), prefix/sequential ($a \rightarrow P$), choice ($P+Q$)
    - o Axioms: indemnity ($\emptyset \| P = P$), commutativity ($P+Q=Q+P$, $P \| Q = Q \| P$)
  - ➢ Manipulate processes by manipulating expressions
- ➢ Parallel programming languages
  - ➢ CSP-based [Occam/Transputer, Handle-C]

State-Based Models

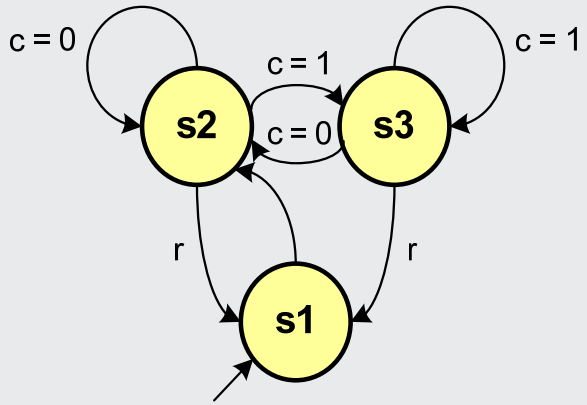
- Sequence and reactivity (control flow)

- Explicit enumeration of computational states
  - ✓ State represents captured history
- Explicit flow of control
  - ✓ Transitions in reaction to events

- Stepwise operation of a machine
  - ✓ Cycle-by-cycle hardware behavior
  - ✓ Finite number of states
- ➢ Formal analysis
  - ✓ Reachability, equivalence

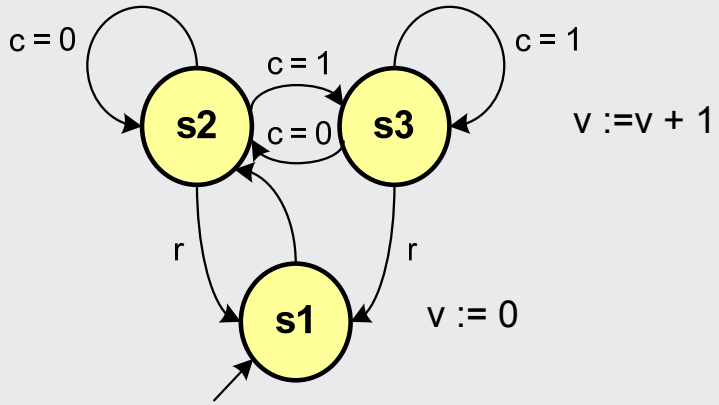


Finite State Machines



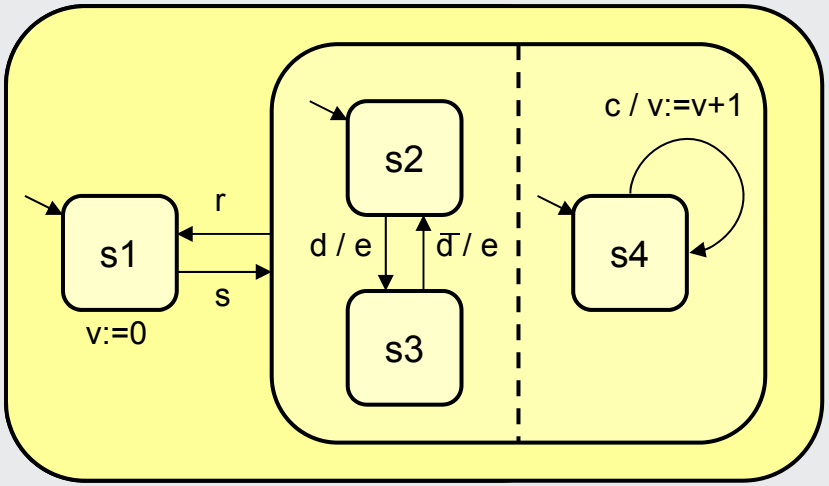
- Finite State Machine (FSM)
  - ✓ Basic model for describing control and automata
    - Sequential circuits
  - ✓ States *S*, inputs/outputs *I/O*, and state transitions
    - FSM: <S, I, O, f, h>
    - Next state function f: S x I → S
  - ✓ Output function *h*
    - Mealy-type (input-based), h: S x I → O
    - Moore-type (state-based), h: S → O

Finite State Machines with Data



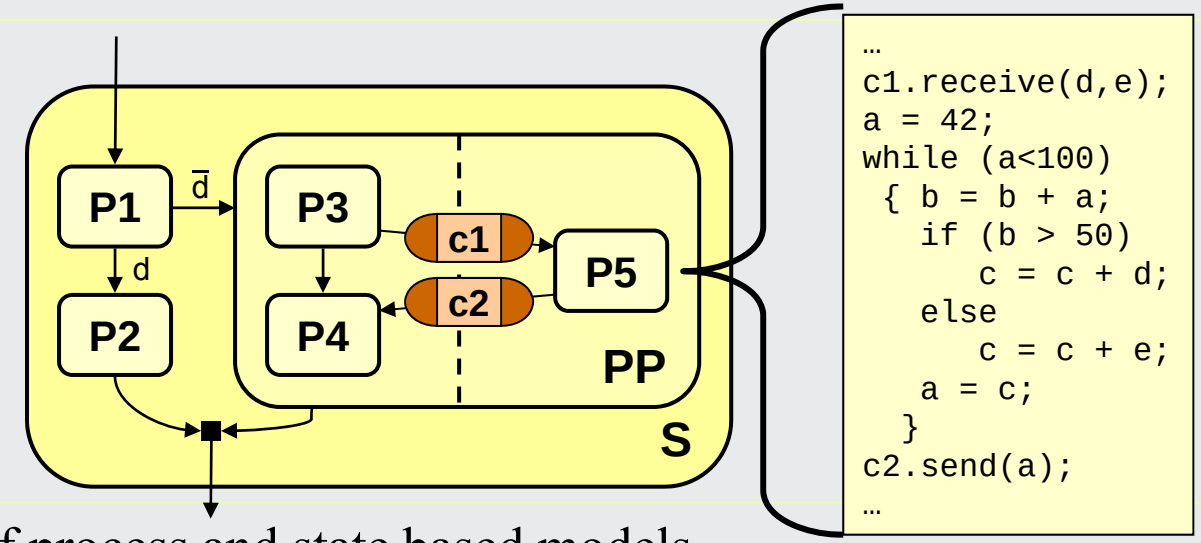
- Finite State Machine with Data (FSMD)
  - ✓ Computation as control and expressions
    - o Controller and datapath of RTL processors
  - ✓ FSM plus variables V
    - o FSMD: $<S, I, O, V, f, h>$
    - o Next state function f: $S \times V \times I \rightarrow S \times V$
    - o Output function h: $S \times V \times I \rightarrow O$

Hierarchical & Concurrent State Machines



- Superstate FSM with Data (SFSMD)
  - ✓ Hierarchy to organize and reduce complexity
    - o Superstates that contain complete state machines each
    - o Enter into one and exit from any substate
- Hierarchical Concurrent FSM (HCFSM)
  - ✓ Hierarchical (OR) and parallel (AND) state composition
  - ✓ Communication through variables, signals and events
  - ✓ Graphical notation [StateCharts]
    - o Lock-step concurrent execution

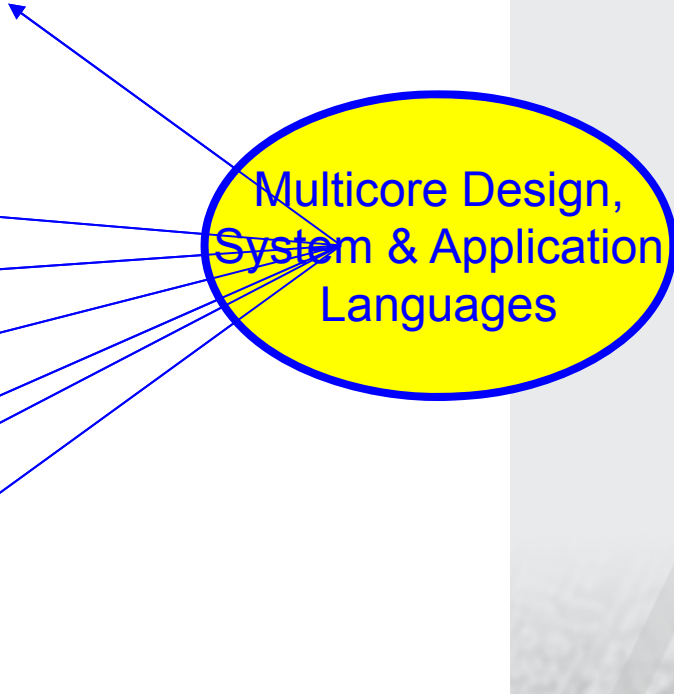
Process State Machines



- Sound combination of process and state based models
  - ✓ Asynchronous concurrent HCFSM execution [UML StateDiagram]
    - o Explicit event queues, deadlock analysis [PetriNet]
  - ✓ Globally asynchronous, locally synchronous (GALS) composition
    - o Co-design Finite State Machines (CFSM) [Polis]
  - ✓ Leaf states are imperative processes
    - o Program State Machine (PSM) [SpecSyn]
- Processes and abstract channels
  - ✓ Computation &communication
    - o Process State Machine (PSM) [SpecC]

Programming and Design Languages

- *Applicative (functional) languages* (revised 1998)
- *Concurrent, distributed, and parallel languages*
- *Constraint and logic languages* (new)
- *Data-flow languages*
- *Design languages*
- *Extensible languages*
- *Macro and assembly languages*
- *Microprogramming languages* (retired '98)
- *Multiparadigm languages* (new)
- *Nondeterministic languages* (retired '98)
- *Nonprocedural languages* (retired '98)
- *Object-oriented languages*
- *Very high-level languages*
- *Specialized application languages*



Languages

- Represent a model in machine-readable form
  - ✓ Apply algorithms and tools
- Syntax defines grammar
  - ✓ Possible strings over an alphabet
  - ✓ Textual or graphical
- Semantics defines meaning
  - ✓ Mapping of strings to an abstract state machine model
    - o Operational semantics
  - ✓ Mapping of strings into a mathematical domain (e.g. functions)
    - o Denotational semantics
- Semantic model vs. MoC vs. design model instance
  - ✓ Basic semantic models can represent many MoCs (e.g. FSMs in C)
  - ✓ MoCs can be represented in different languages

Design Languages

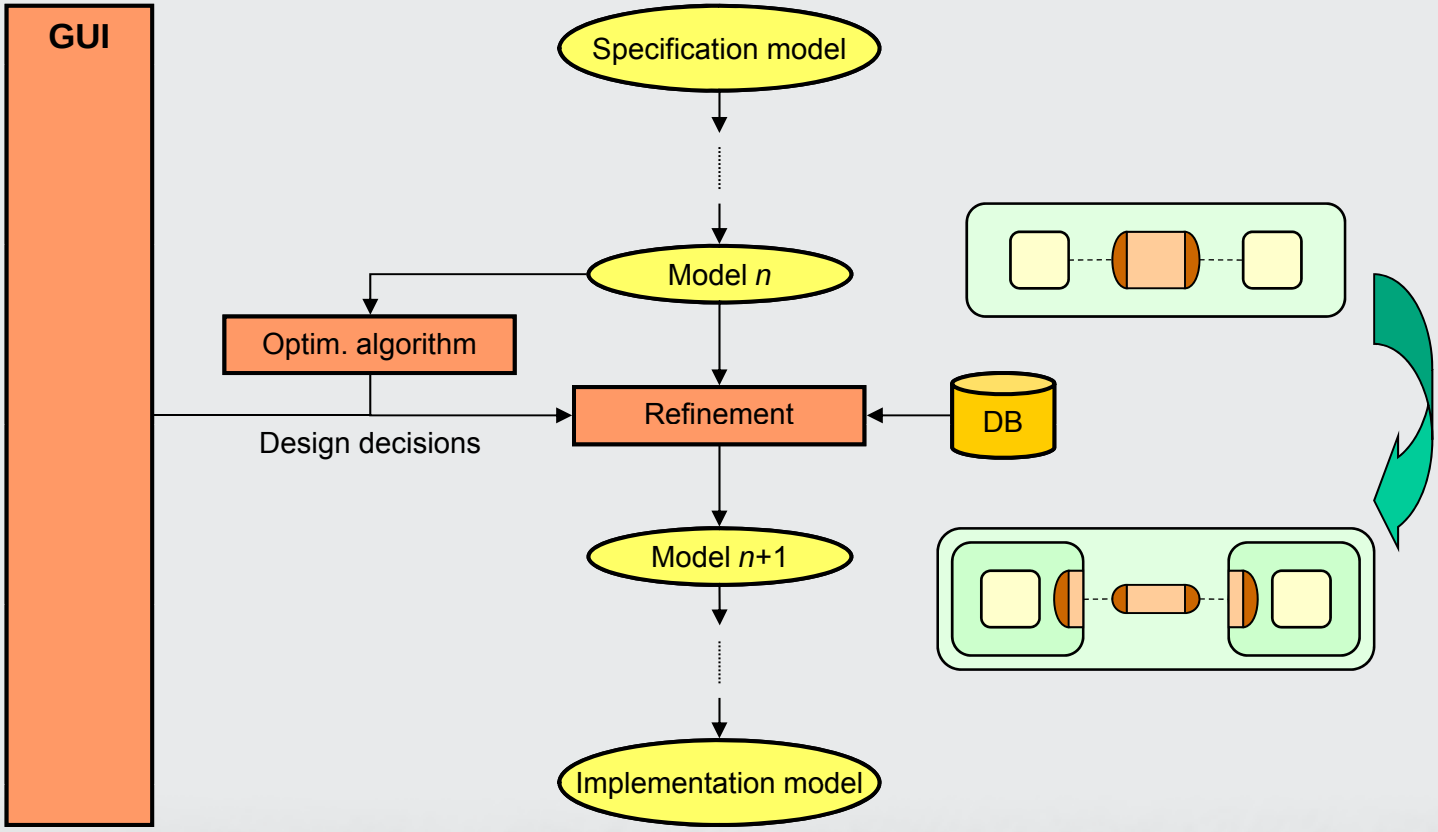
- Netlists
  - ✓ Structure only: components and connectivity
    - o Gate-level [EDIF], system-level [SPIRIT/XML]
- Hardware description languages (HDLs)
  - ✓ Event-driven behavior: signals/wires, clocks
  - ✓ Register-transfer level (RTL): boolean logic
    - o Discrete event [VHDL, Verilog]
- System-level design languages (SLDLs)
  - ✓ Software behavior: sequential functionality/programs
    - o C-based, event-driven [SpecC, SystemC, SystemVerilog]

System Modeling

- Basis of any design flow and design automation
  - ✓ Inputs and outputs of design steps
    - o Capability to capture complex systems
  - ✓ Models at varying levels of abstraction
    - o Level and granularity of implementation detail
- Design models as an abstraction of a design instance
  - ✓ Representation of some aspect of reality
    - o Virtual prototyping for analysis and validation
  - ✓ Specification for further implementation/synthesis
    - o Describe desired functionality
  - ✓ Documentation & specification
    - o Represent design decisions that have been made
    - o Abstraction to hide details that are not relevant or not yet known

Design Process

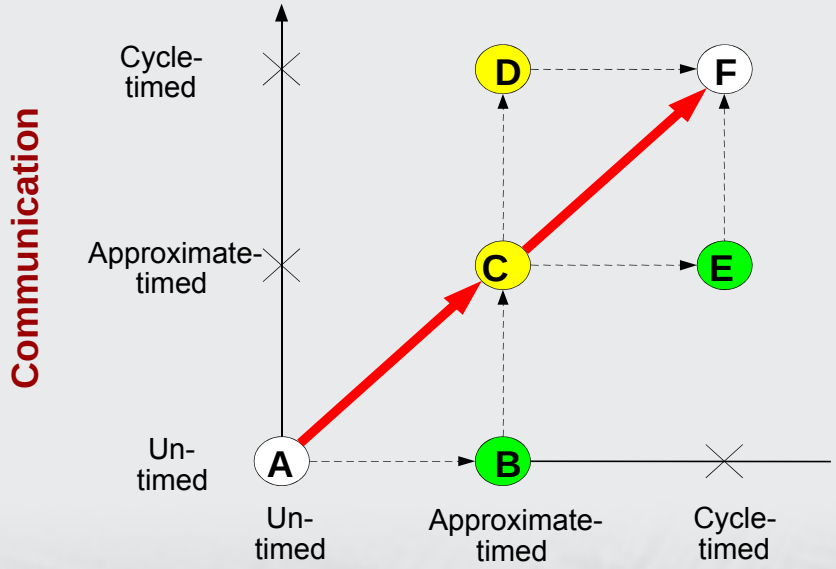
- From specification to implementation



- Successive, stepwise model refinement
- Layers of implementation detail

Abstraction Levels

- Abstraction based on level of detail & granularity
  - ✓ Computation and communication
- System design flow
  - ✓ Path from model A to model F

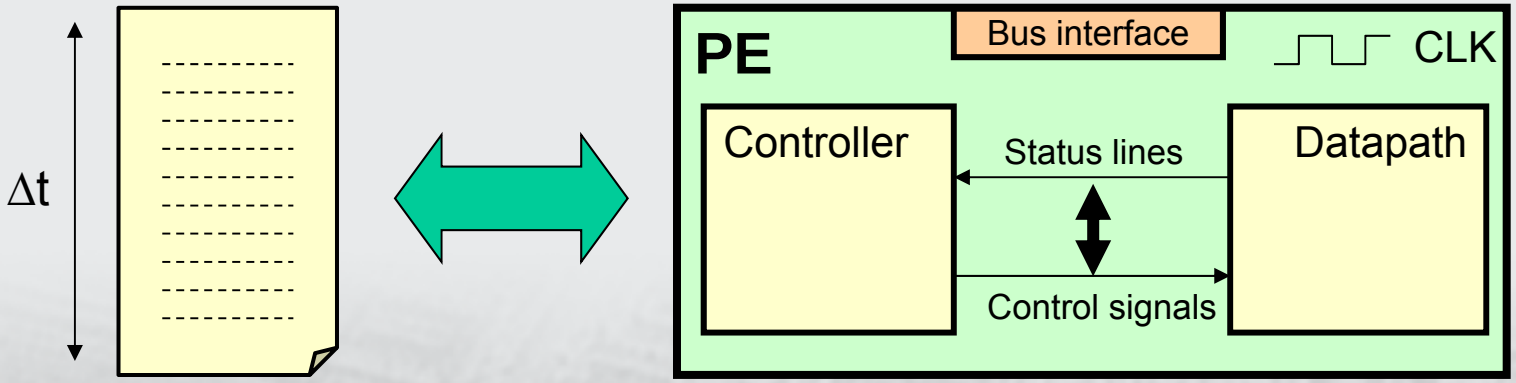


A. System specification model
B. Timed functional model
C. Transaction-level model (TLM)
D. Bus cycle-accurate model (BCAM)
E. Computation cycle-accurate model (CCAM)
F. Cycle-accurate model (CAM)

- Design methodology and modeling flow
  - ✓ Set of models and transformations between models

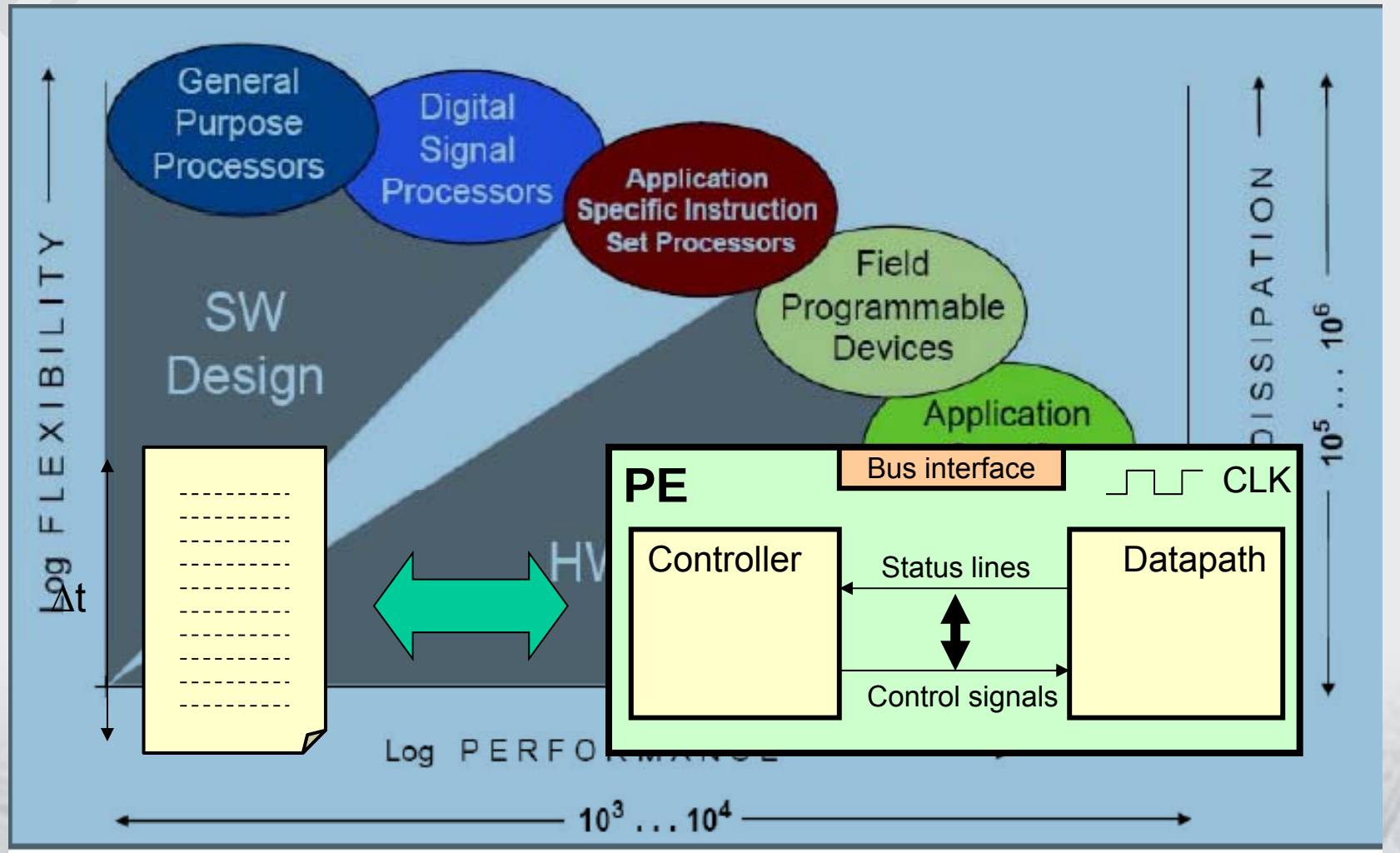
Processors

- Basic system component is a *processor*
  - ✓ Programmable, general-purpose software processor (CPU)
  - ✓ Programmable special-purpose processor (e.g. DSPs)
  - ✓ Application-specific instruction set processor (ASIP)
  - ✓ Custom hardware processor



- Functionality and timing

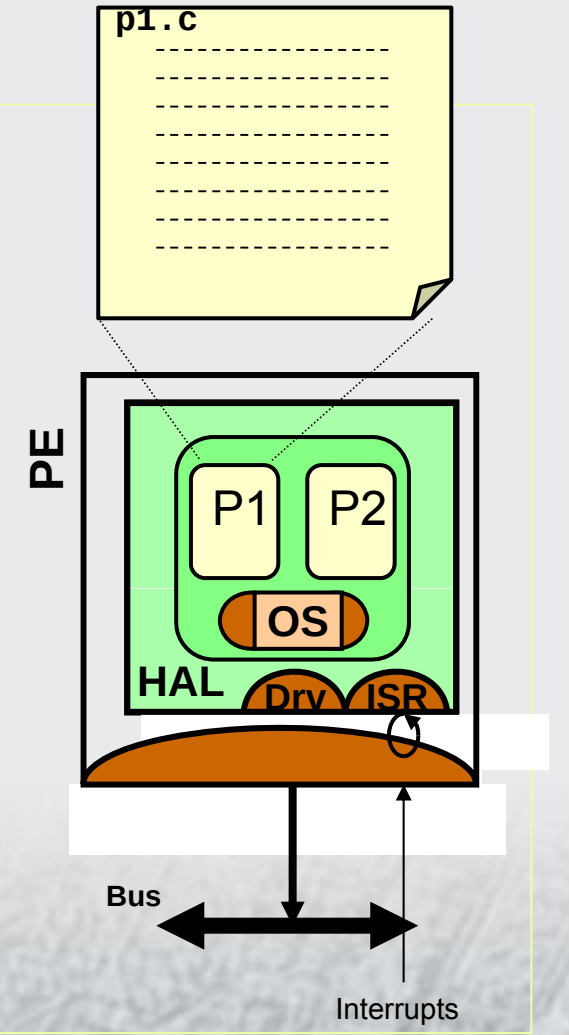
Processors



Source: T. Noll, RWTH Aachen, via R. Leupers, "From ASIP to MPSoC", Computer Engineering Colloquium, TU Delft, 2006

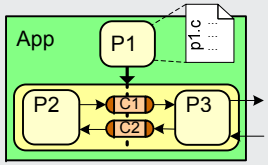
Processor Modeling

- Application modeling
  - ✓ Native process execution (C code)
  - ✓ Back-annotated execution timing
- Processor modeling
  - ✓ Operating system (OS)
    - o Real-time multi-tasking (RTOS)
    - o Bus drivers (C code)
  - ✓ Hardware abstraction layer (HAL)
    - o Interrupt handlers
    - o Media accesses
  - ✓ Processor hardware
    - o Bus interfaces (I/O state machines)
    - o Interrupt suspension and timing

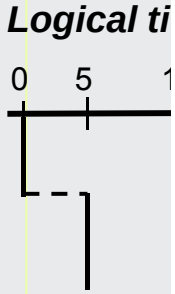


Application Layer

- High-level, abstract programming model
  - ✓ Hierarchical process graph
    - o ANSI C leaf processes
    - o Parallel-serial composition
  - ✓ Abstract inter-process communication
    - o Channels
    - o Shared variables
- Timed simulation of application functionality (SLDL)
  - ✓ Back-annotate timing
    - o Estimation or measurement (trace, ISS)
    - o Function or basic block level granularity
  - ✓ Execute natively on simulation host
    - o Discrete event simulator
    - o Fast, native compiled simulation

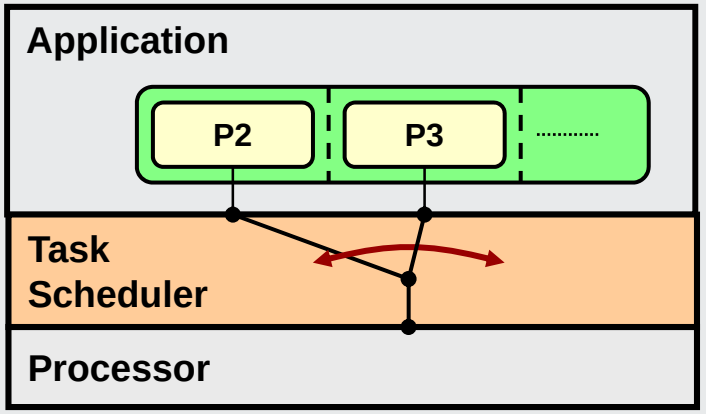



```
process p1()
{
  ...
  waitfor(5);
  ...
}
```

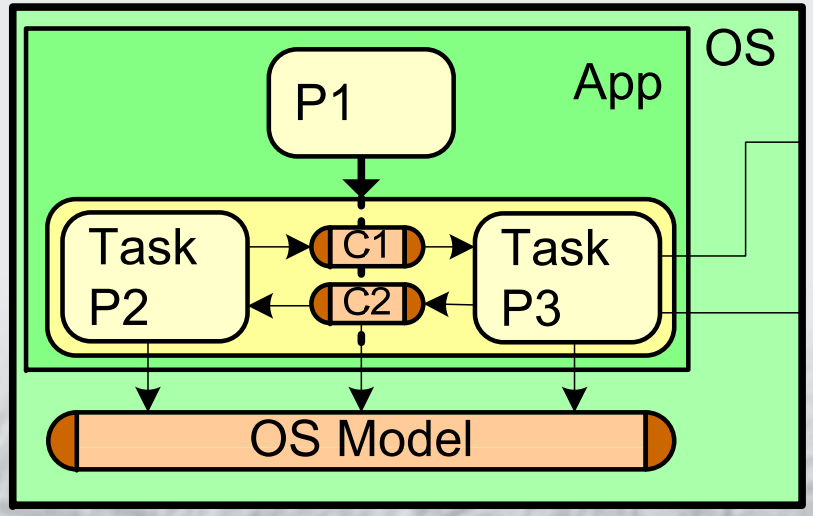


Operating System Layer

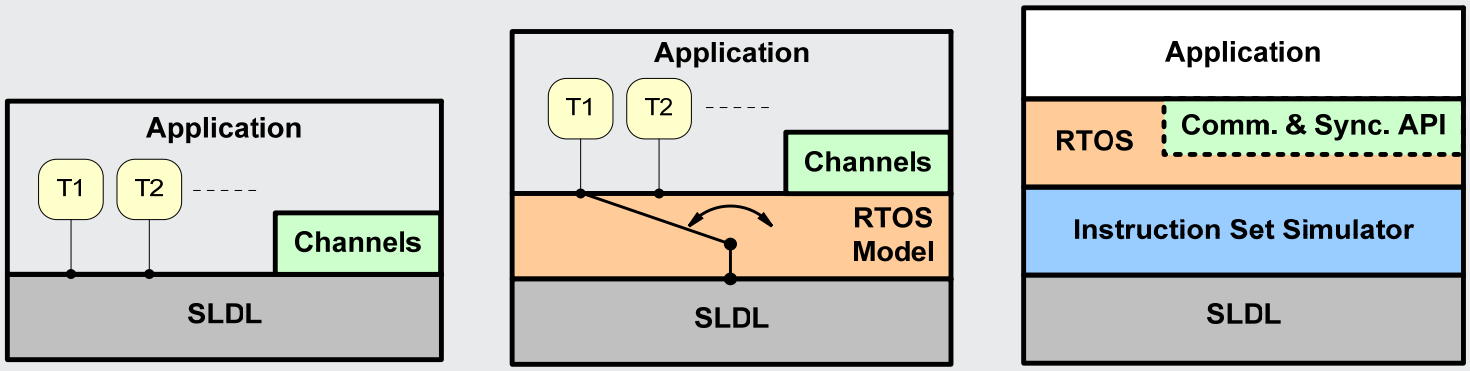
- Scheduling
  - ✓ Group processes into tasks
    - o Static scheduling
  - ✓ Schedule tasks
    - o Dynamic scheduling, multitasking
    - o Preemption, interrupt handling
    - o Task communication (IPC)



- Scheduling refinement
  - ✓ Flatten hierarchy
  - ✓ Reorder behaviors
- OS refinement
  - ✓ Insert OS model
  - ✓ Task refinement
  - ✓ IPC refinement



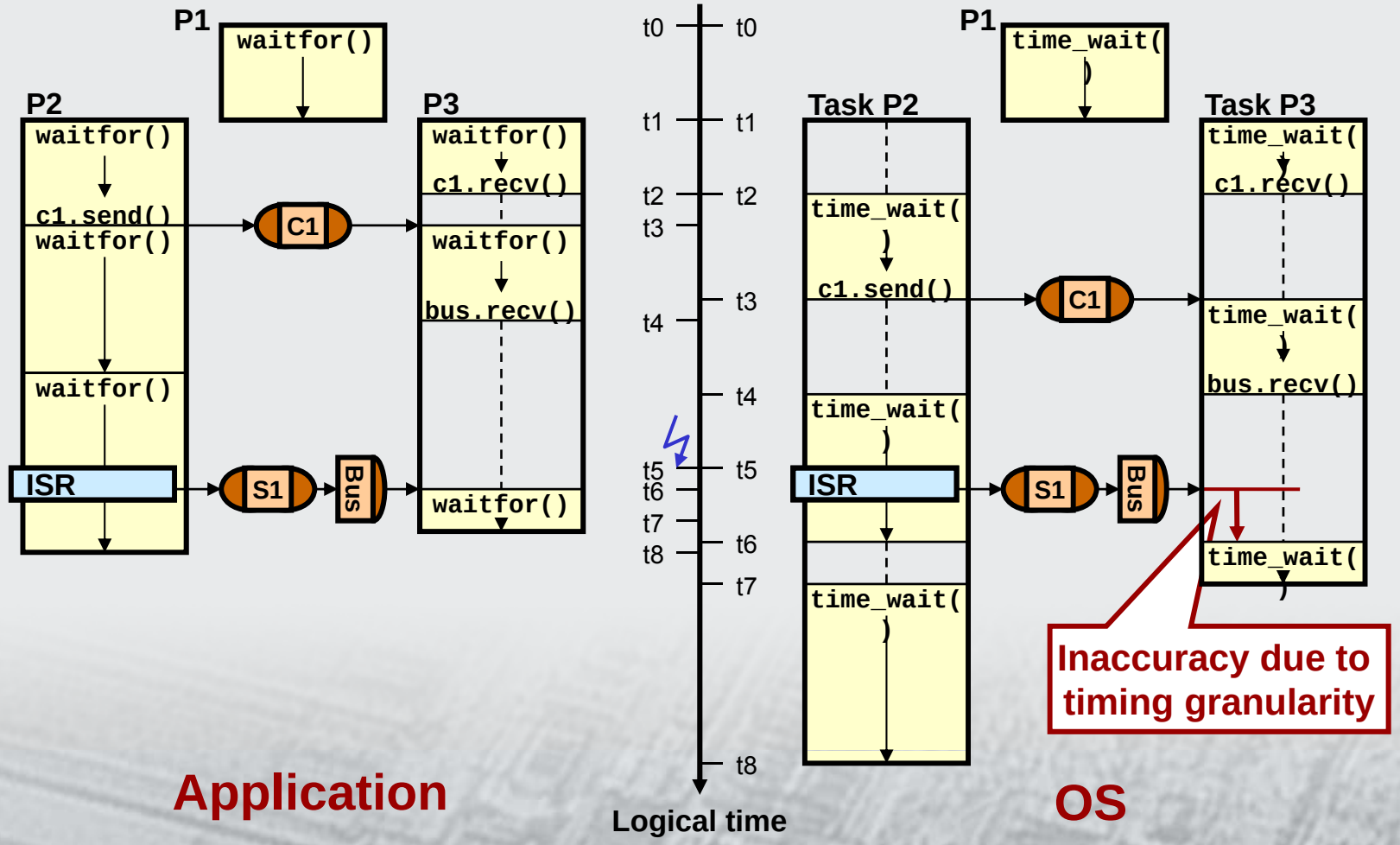
OS Modeling



Specification *TLM* Implementation

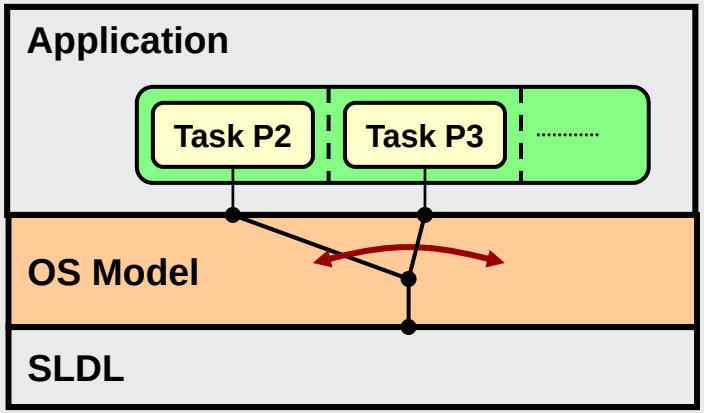
- High-level RTOS abstraction
  - ✓ Specification is fast, but inaccurate
    - o Native execution, concurrency model
  - ✓ Traditional ISS-based validation infeasible
    - o Accurate but slow (esp. in multi-processor context), requires full binary
  - ✓ Model of operating system
    - o High accuracy but small overhead at early stages
    - o Focus on key effects, abstract unnecessary implementation details
    - o Model all concepts: Multi-tasking, scheduling, preemption, interrupts, IPC

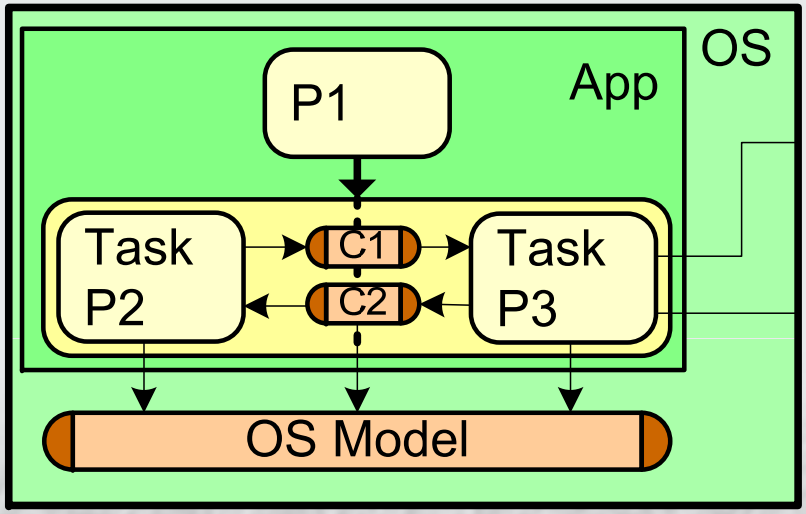
Simulated Dynamic Behavior



Operating System Layer

- OS model
  - ✓ On top of standard SLDL
  - ✓ Wrap around SLDL primitives, replace event handling
    - o Block all but active task
    - o Select and dispatch tasks
  - ✓ Target-independent, canonical API
    - o Task management
    - o Channel communication
    - o Timing and all events



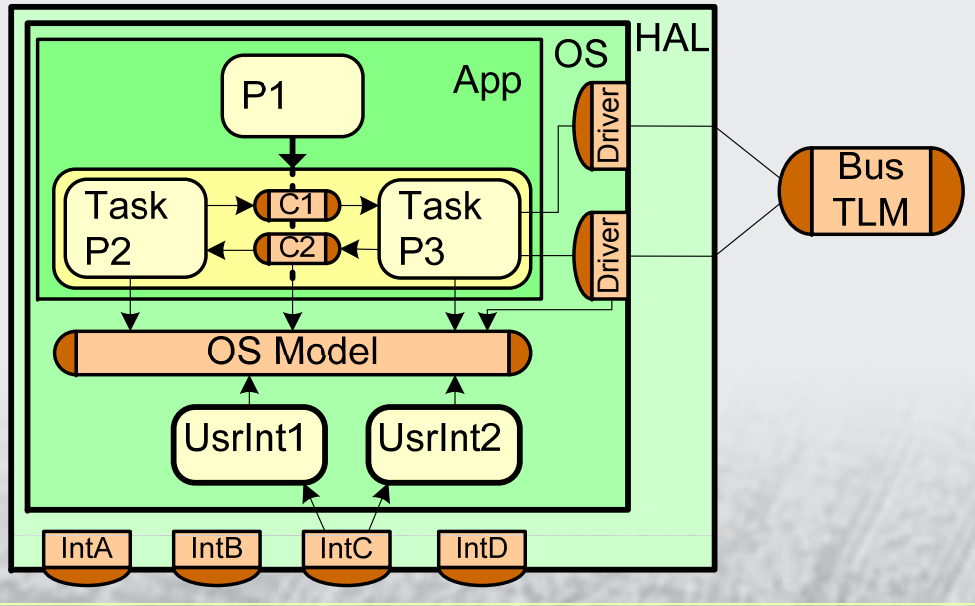


Hardware Abstraction Layer

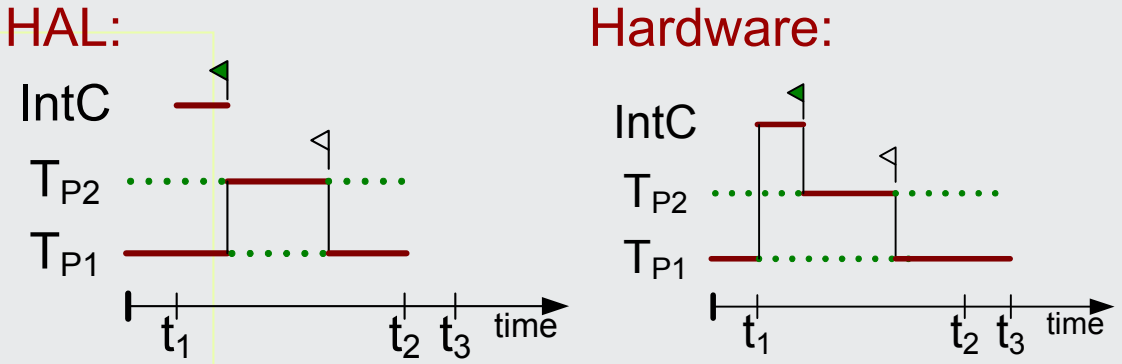
- External communication
  - ✓ Software Drivers
    - o Presentation, session, network communication layers
    - o Synchronization (interrupts)
  - ✓ Hardware/software boundary
    - o Low-level HW access
    - o Bus drivers and interrupt handlers
    - o Canonical HW/SW interface
  - ✓ External interface
    - o Bus transactions (TLM)
    - o Interrupt trigger

App.
```
sample.send(v1);
```

Driver
```
void send(…) {
  intr.receive();
  bus.masterWrite(0xA000,
                  &tmp,
                  len);
}
```



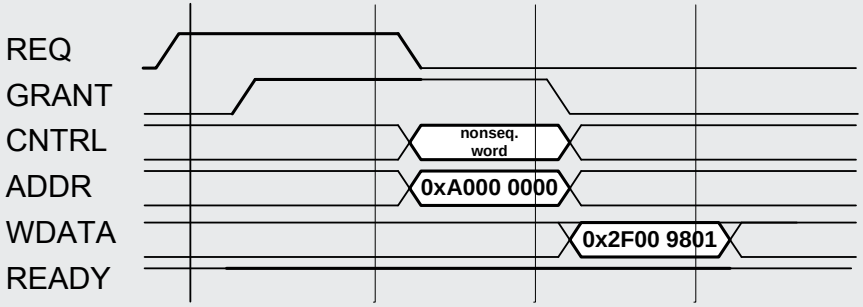
Hardware Layer (1)

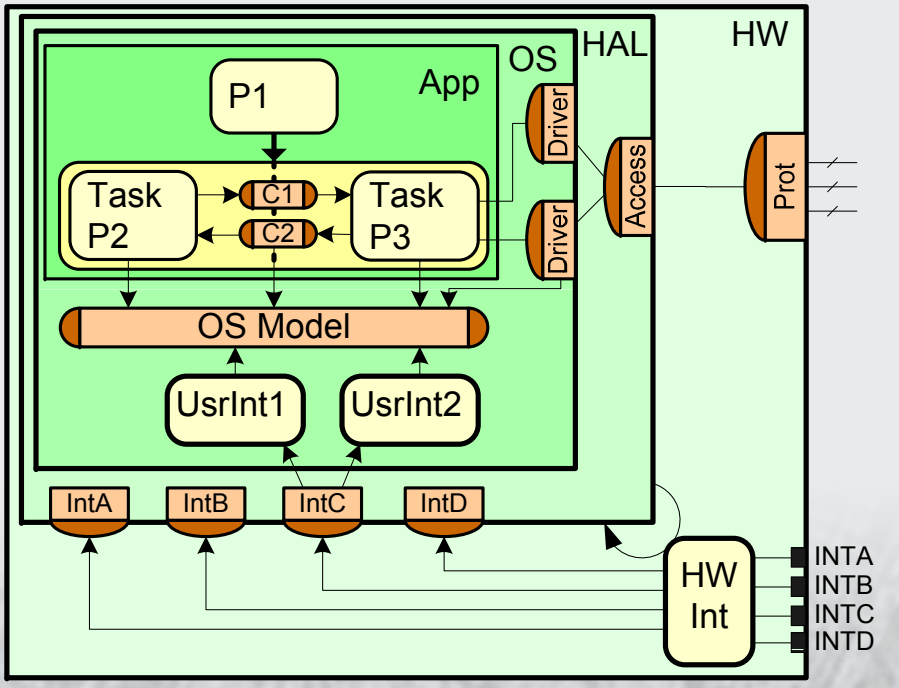


- Processor TLM
  - ✓ HW interrupt handling
    - o Interrupt logic
      - – Suspend user code
    - o Interrupt scheduling
      - – Priority, nesting
  - ✓ Peripherals
    - o Interrupt controller
    - o Timers
  - ✓ TLM bus model
    - o Bus transactions

Hardware Layer (2)

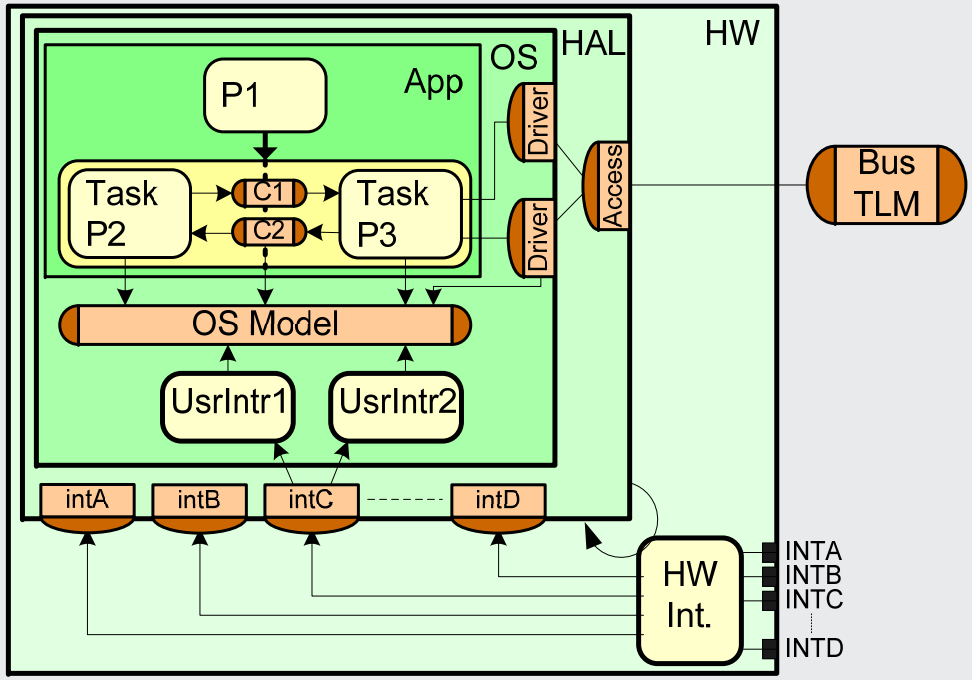
- Bus-functional model (BFM)
  - ✓ Pin-accurate processor model
    - o Timing-accurate bus and interrupt protocols
  - ✓ Bus model
    - o Pin- and cycle-accurate
    - o Driving and sampling of bus wires





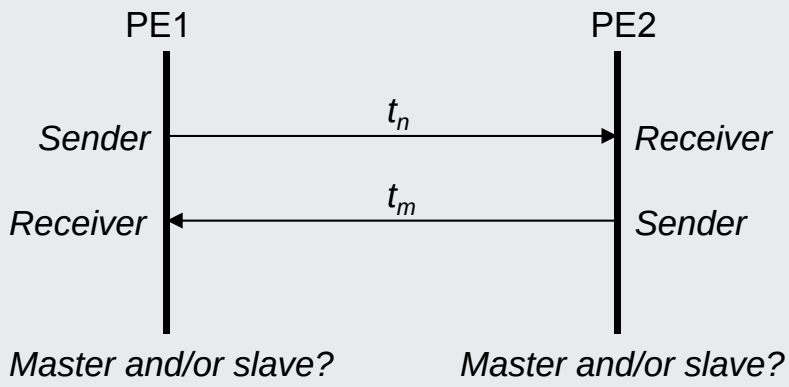
Processor Model

- Processor layers
  - ✓ Application
    - Native C
    - Back-annotated timing
  - ✓ Operating system
    - OS model
  - ✓ Hardware abstraction
    - Middleware, Firmware
  - ✓ Processor hardware
    - Bus I/F
    - Interrupts, suspension



Features	
Target approx. computation timing	Appl.
Task mapping, dynamic scheduling Task communication, synchronization	OS
Interrupt handlers, low level SW drivers	HAL
HW interrupt handling, int. scheduling	HW-TLM
Cycle accurate communication	HW-BFM
Cycle accurate computation	BFM - ISS

Bus



- For each transaction between two communication partners
  - ✓ 1 sender, 1 receiver
  - ✓ 1 master (initiator), 1 slave (listener)
- Any combination of master/slave, sender/receiver
  - ✓ Master/Slave bus
    - o Statically fixed master/slave assignments for each PE pair
    - o PEs can be masters, slaves or both (dual-port)
  - ✓ Node-based bus (e.g. Ethernet, CAN):
    - o Sender is master, receiver is slave
- Reliable (loss-less, error-free)?

Communication Primitives

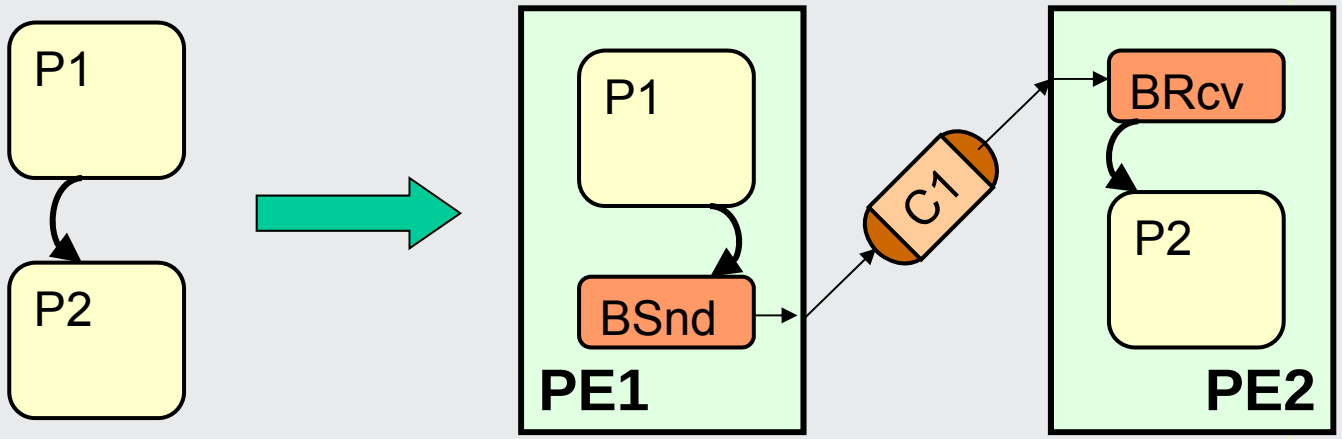
- Events, transitions
  - ✓ Pure control flow, no data
- Shared variables
  - ✓ No control flow, no synchronization
- Synchronous message passing
  - ✓ No buffering, two-way control flow
- Asynchronous message passing
  - ✓ Only control flow from sender to receiver guaranteed
  - ✓ May or may not use buffers (implementation dependent)
- Queues
  - ✓ Fixed, defined queue length (buffering)
- Complex channels
  - ✓ Semaphores, mutexes
- Reliable communication primitives (lossless, error-free)

Communication (OSI Reference Model)

Layer	Semantics	Functionality	Implementation	OSI
Application	Channels, variables	Computation	Application	7
Presentation	End-to-end typed messages	Data formatting	OS	6
Session	End-to-end untyped messages	Synchronization, Multiplexing	OS	5
Transport	End-to-end data streams	Packeting, Flow control	OS	4
Network	End-to-end packets	Subnet bridging, Routing	OS	3
Link	Point-to-point logical links	Station typing, Synchronization	Driver	2b
Stream	Point-to-point control/data streams	Multiplexing, Addressing	Driver	2b
Media Access	Shared medium byte streams	Data slicing, Arbitration	HAL	2a
Protocol	Media (word/frame) transactions	Protocol timing	Hardware	2a
Physical	Pins, wires	Driving, sampling	Interconnect	1

Application Layer (1)

- Synchronization
  - ✓ Synthesize control flow

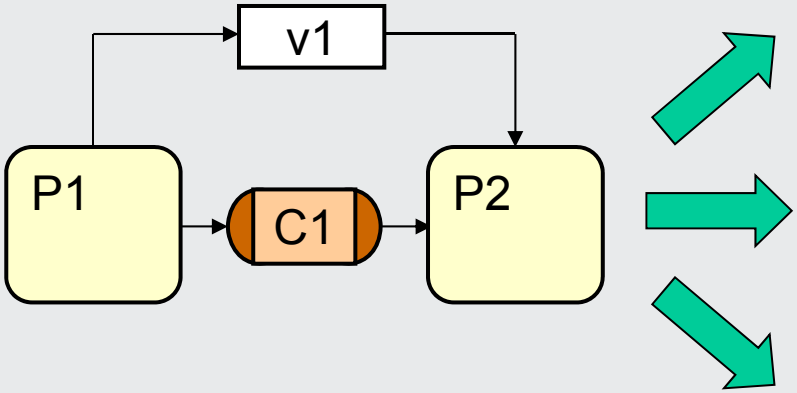


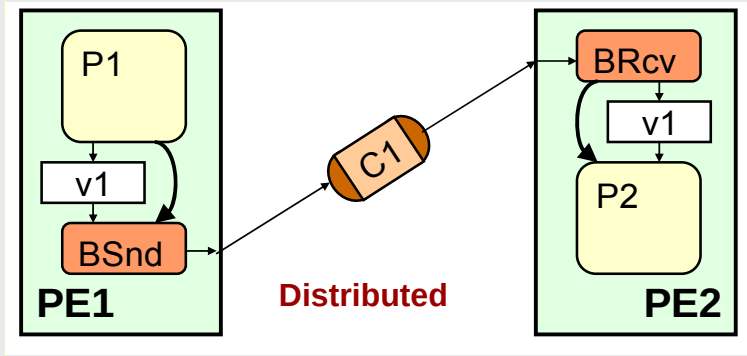
Parallel processes plus synchronization events

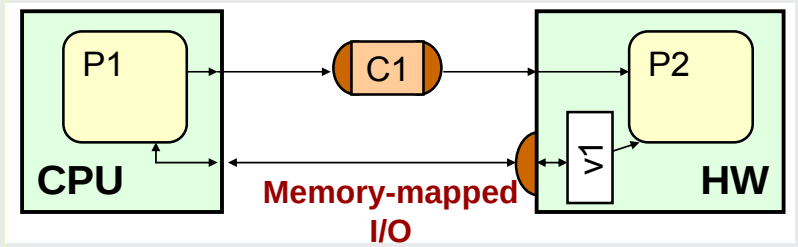
  - ✓ Implement sequential transitions across parallel components

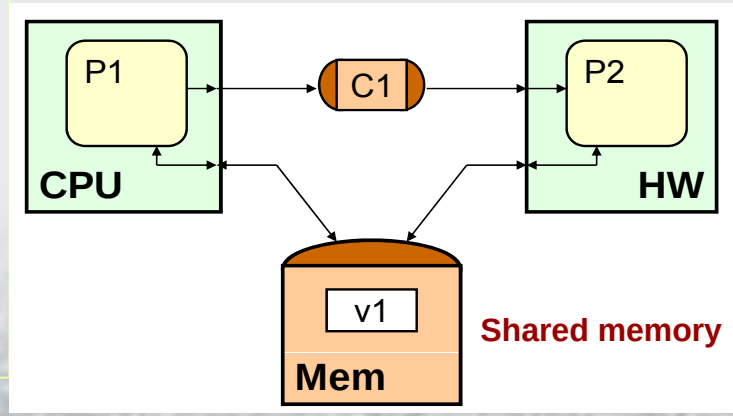
Application Layer (2)

- Storage
  - ✓ Shared variable mapping to memories





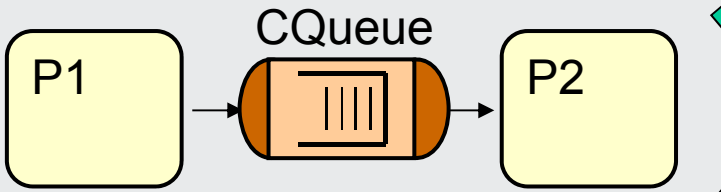


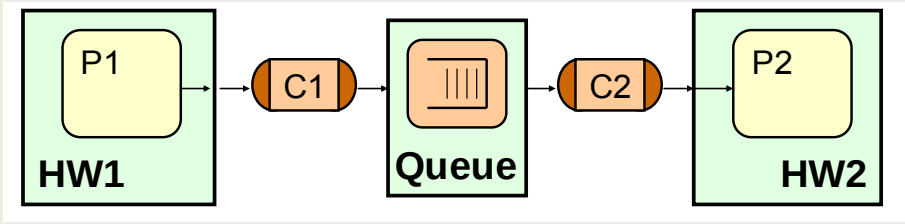


  - ✓ Map global storage to local memories

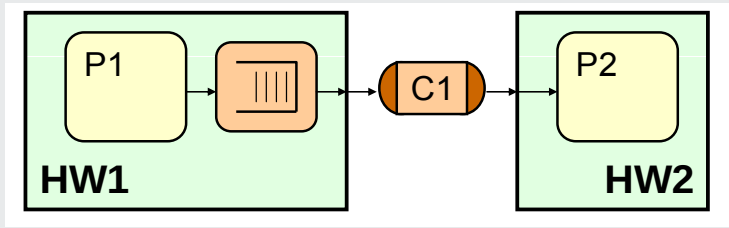
Application Layer (3)

- Channels
  - ✓ Complex channel synthesis





Dedicated hardware

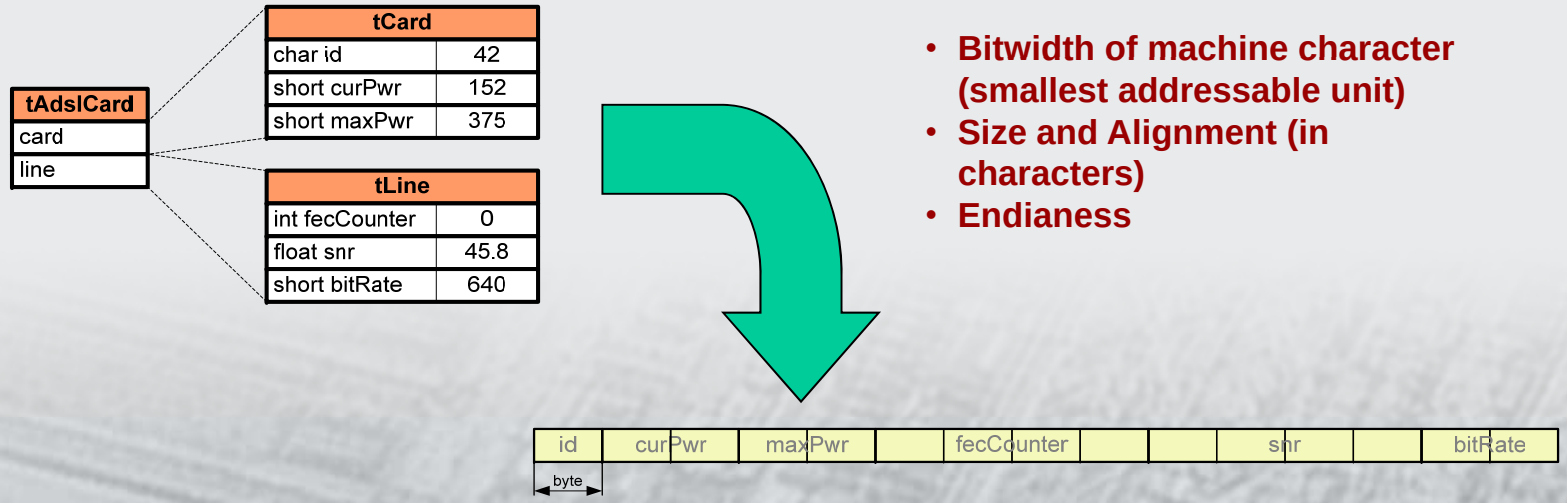


Additional process

  - ✓ Client-server implementation
    - o Server process
    - o Remote procedure call (RPC) channels

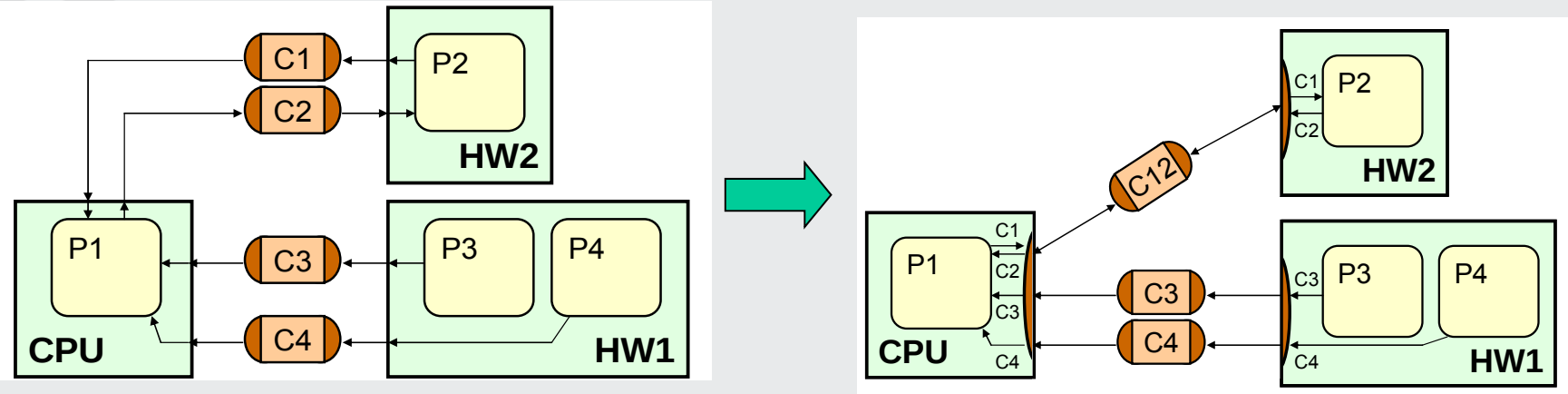
Presentation Layer

- Data formatting
  - ✓ Translate abstract data types into canonical network byte layout
    - o Global network data layout
    - o Shared, optimized layout for each pair of communicating PEs
  - ✓ Convert typed messages into untyped, ordered byte streams
  - ✓ Convert variables into memory byte layout



- **Bitwidth of machine character (smallest addressable unit)**
- **Size and Alignment (in characters)**
- **Endianess**

Session Layer

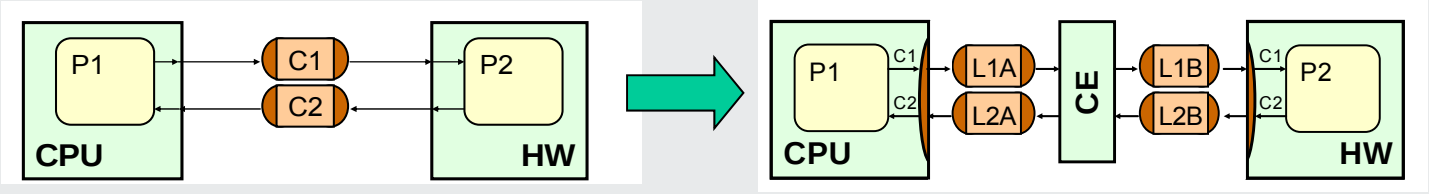


- Channel merging
  - ✓ Merge application channels into a set of untyped end-to-end message streams
    - o Unconditionally merge sequential channels
    - o Merge concurrent channels with additional session ID (message header)
  - ✓ Channel selection over end-to-end transports

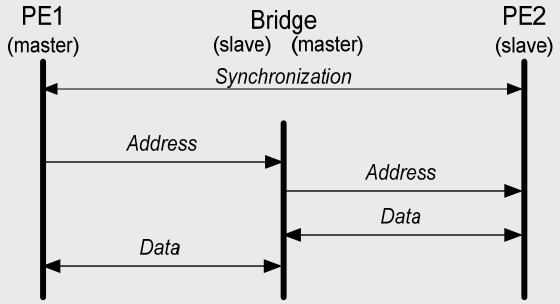
Transport Layer

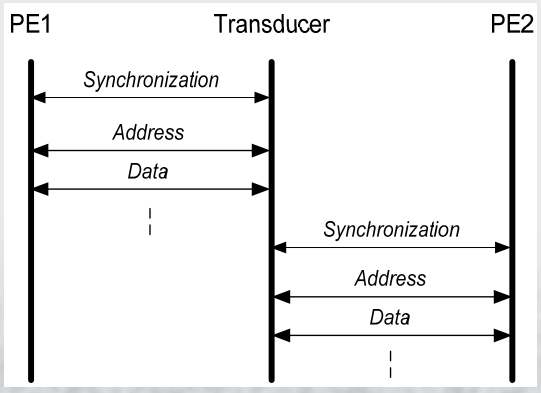
- Packeting and routing
  - ✓ Packetization to reduce buffer sizes
    - o Fixed packet sizes (plus padding)
    - o Variable packet size (plus length header)
  - ✓ Protocol exchanges (ack) to restore synchronicity
    - o synchronous message passing and transducer in the path
  - ✓ Packet switching and identification (logical routing)
    - o Dedicated logical links (defer identification to lower layers)
    - o Network endpoint addressing (plus packet address headers)
  - ✓ Physical routing in case of multiple paths between PEs
    - o Static, predetermined routing (based on connectivity/headers)
    - o Dynamic (runtime) routing

Network Layer



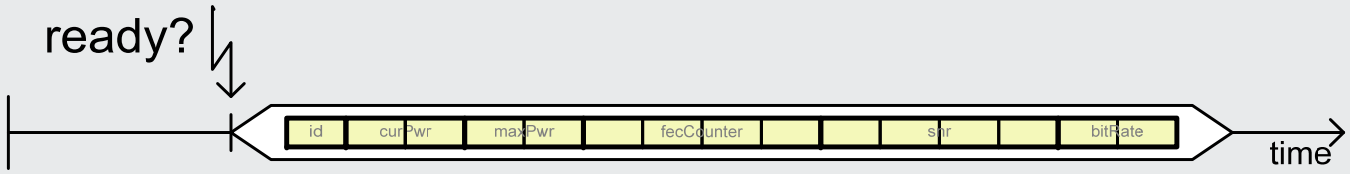
- Split network into subnets
  - ✓ Routing end-to-end over point-to-point links
  - ✓ Insert communication elements (CEs) to connect busses
- Bridges
  - ✓ Transparently connect slave & master sides at protocol level
  - ✓ Bridges maintain synchronicity, no buffers
- Transducers
  - ✓ Store-and-forwarding of data packets between incompatible busses
  - ✓ Intermediate buffering, results in asynchronous communication





Link Layer (1)

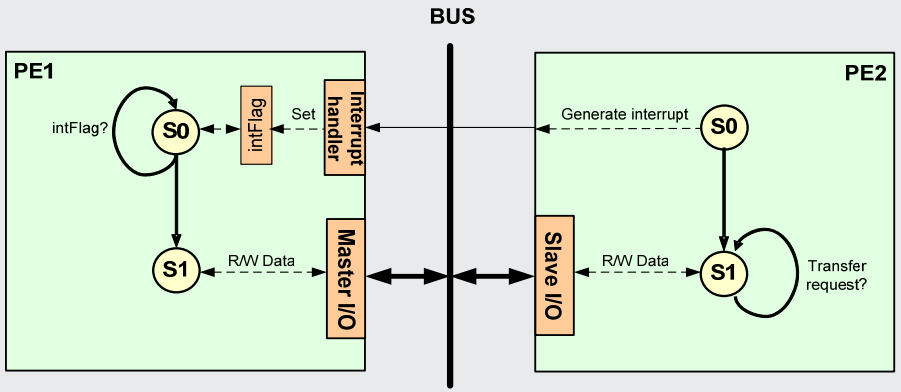
- Synchronization (1)
  - ✓ Ensure slave is ready before master initiates transaction
    - o Always ready slaves (memories and memory-mapped I/O)
    - o Defer to fully synchronized bus protocol (e.g. RS232)
    - o Separate synchronization mechanism



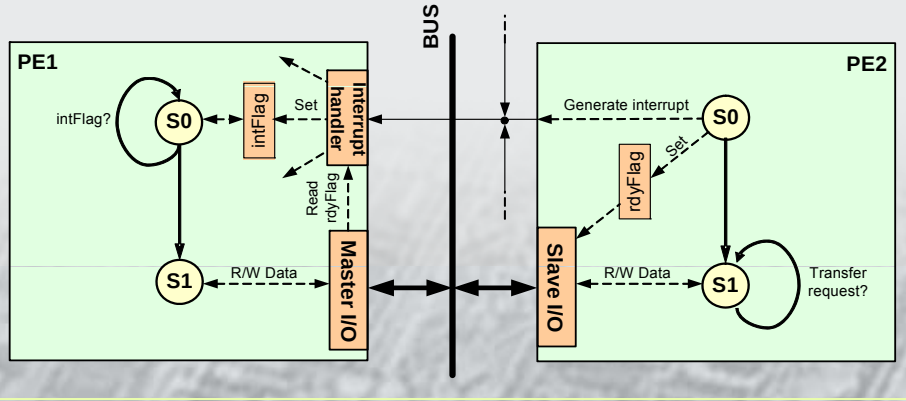
  - ✓ Events from slave to master for master/slave busses
  - ✓ Synchronization packets for node-based busses

Link Layer (2)

- Synchronization
  - ✓ Dedicated interrupts

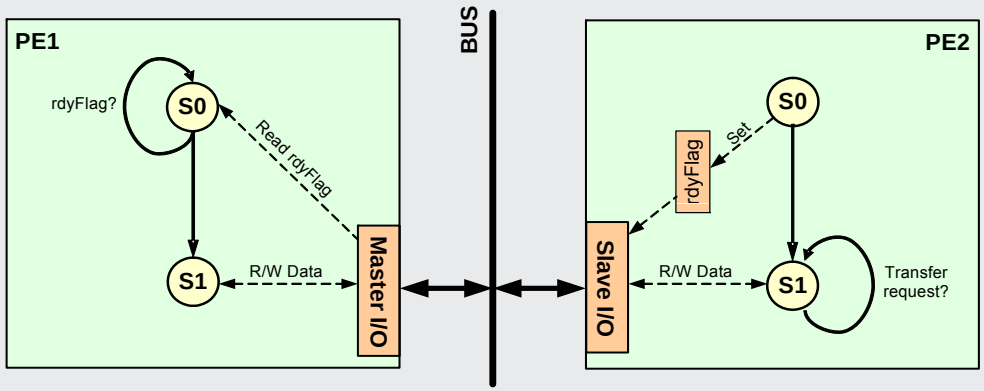


  - ✓ Shared interrupts

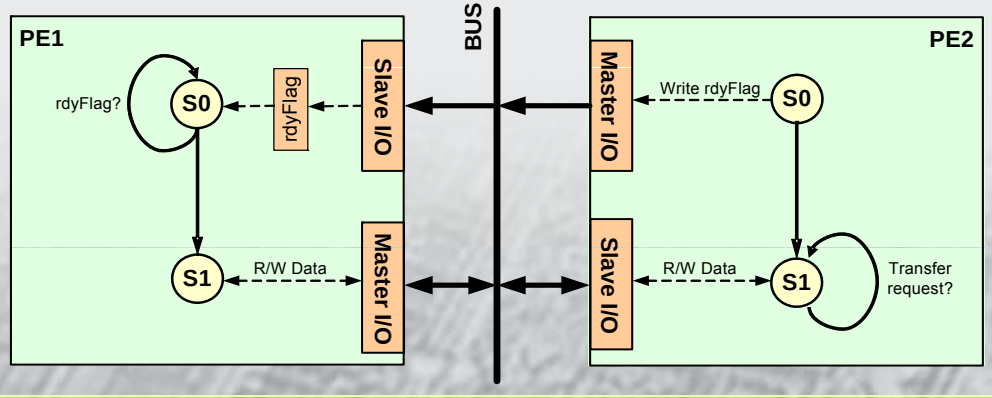


Link Layer (3)

- Synchronization
  - ✓ Slave polling



  - ✓ Flag in master

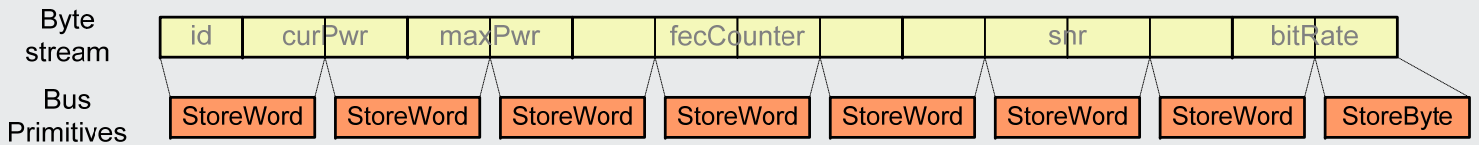


Stream Layer

- Addressing
  - ✓ Multiplexing of links over shared medium
  - ✓ Separation in space through addressing
  - ✓ Assign physical bus addresses to links
    - o Dedicated physical addresses per link
    - o Shared physical addresses plus packet ID/address in packet header

Media Access (MAC) Layer

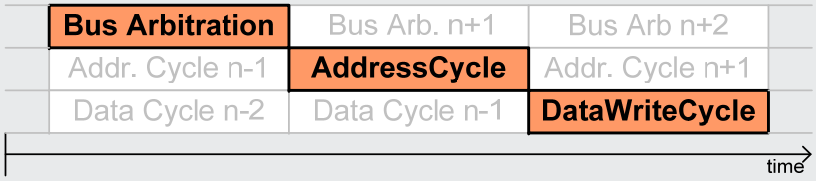
- Data slicing
  - ✓ Split data packets into multiple bus word/frame transactions



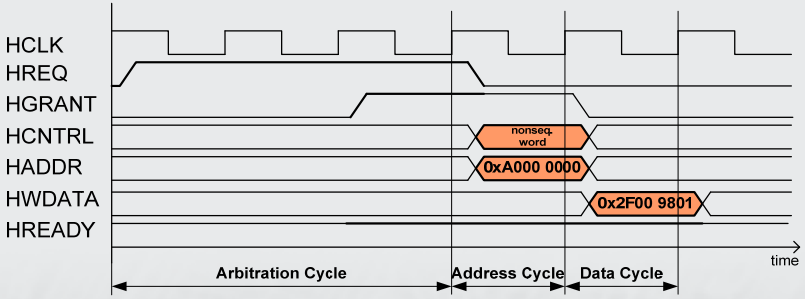
  - ✓ Optimized data slicing utilizing supported bus modes (e.g. burst)
- Arbitration
  - ✓ Separate individual bus transactions in time
    - o Centralized using arbiters
    - o Distributed
  - ✓ Insert arbiter components

Protocol, Physical Layers

- Bus interface
  - ✓ Generate state machines implementing bus protocols
  - ✓ Timing-accurate based on timing diagrams and timing constraints

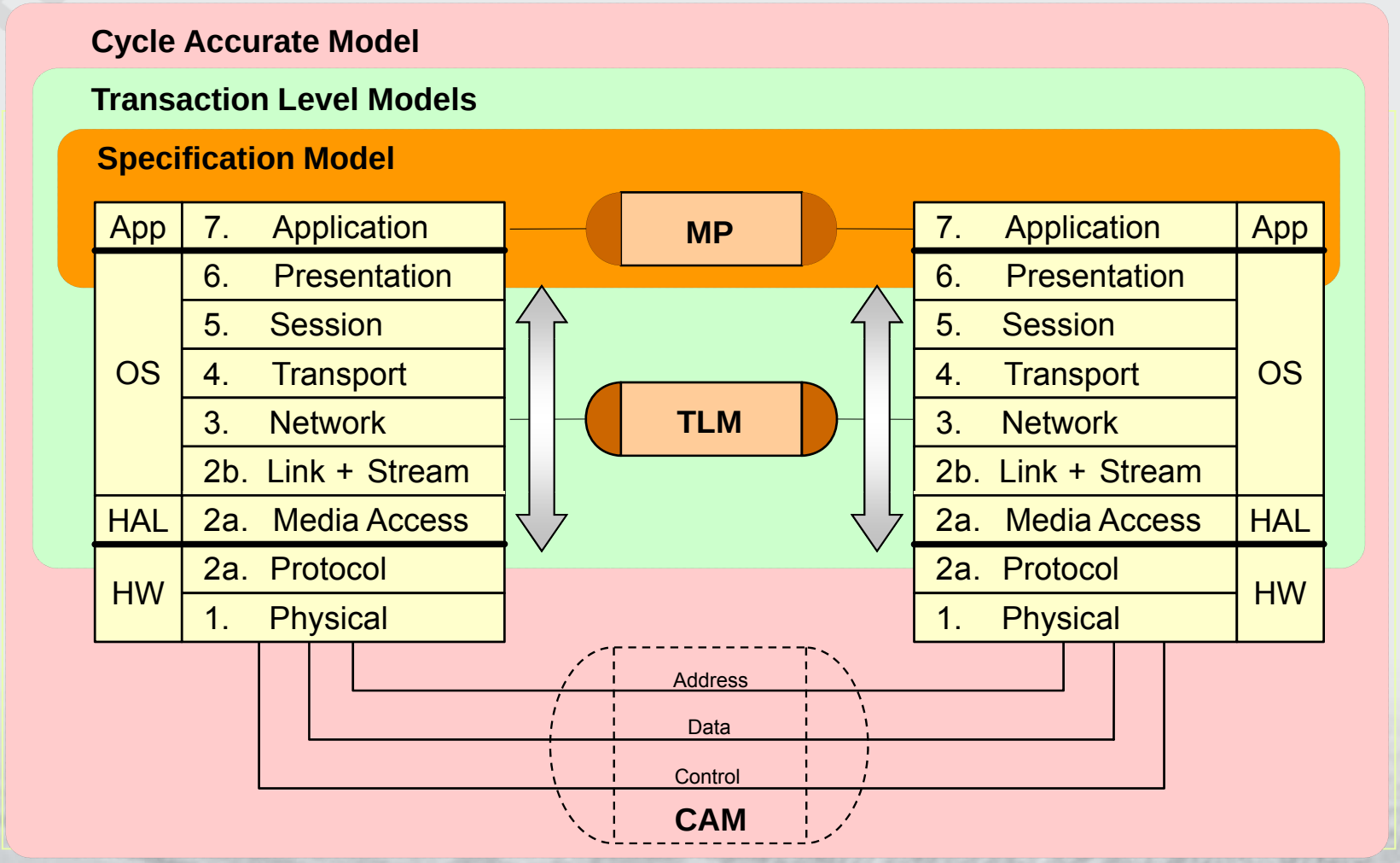


  - ✓ Bus protocol database



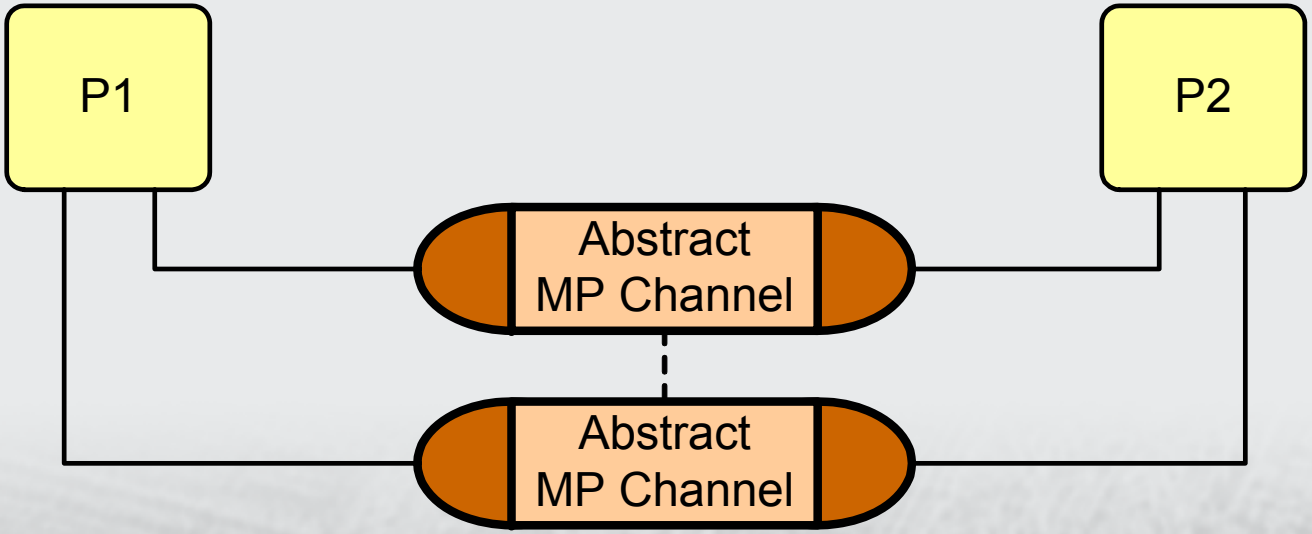
- Port mapping and bus wiring
  - ✓ Connectivity of component ports to bus, interrupt wires/lines
  - ✓ Generate top-level system netlist

System Models

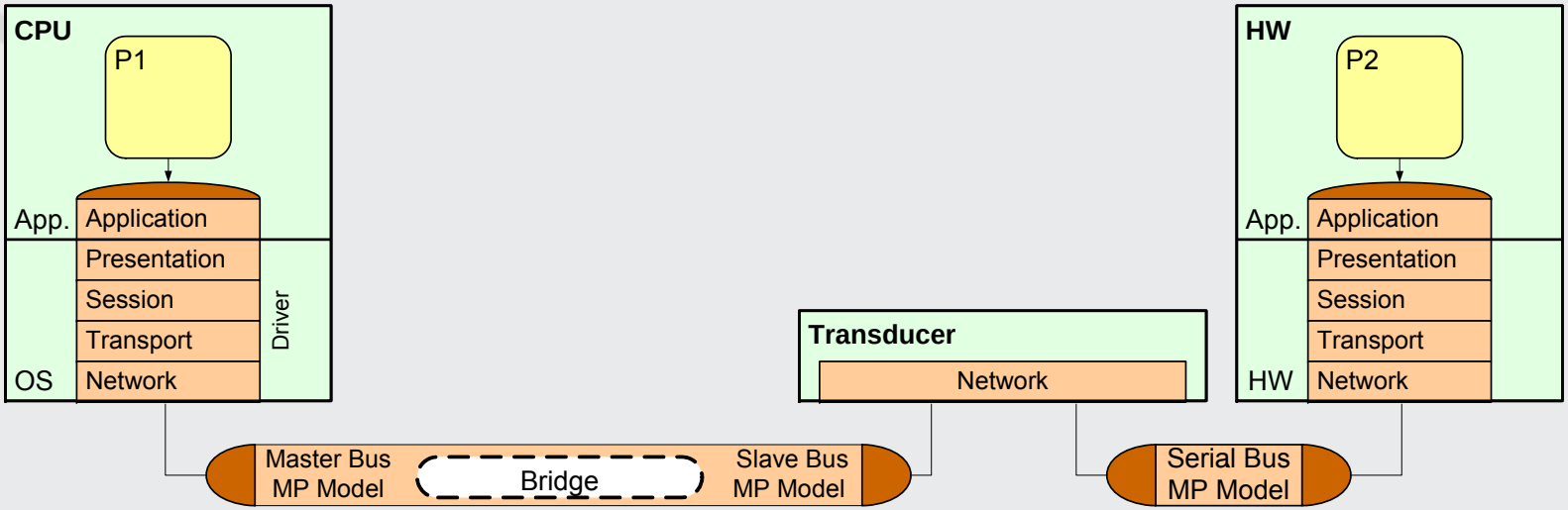


Specification Model

- Abstract, high-level system functionality
  - ✓ Computation
    - o Processes
    - o Variables
  - • Communication
    - – Sync./async. message-passing
    - – Memory interfaces
    - – Events

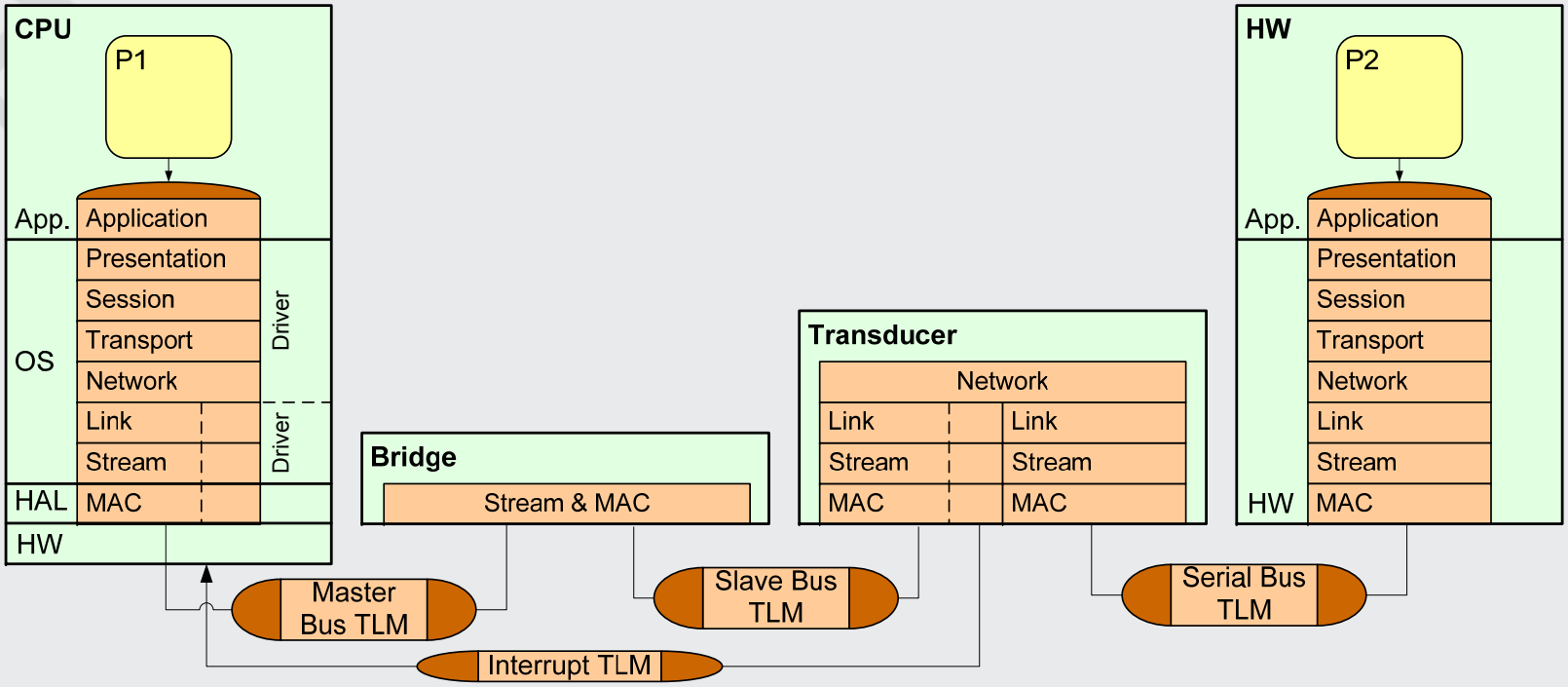


Network TLM



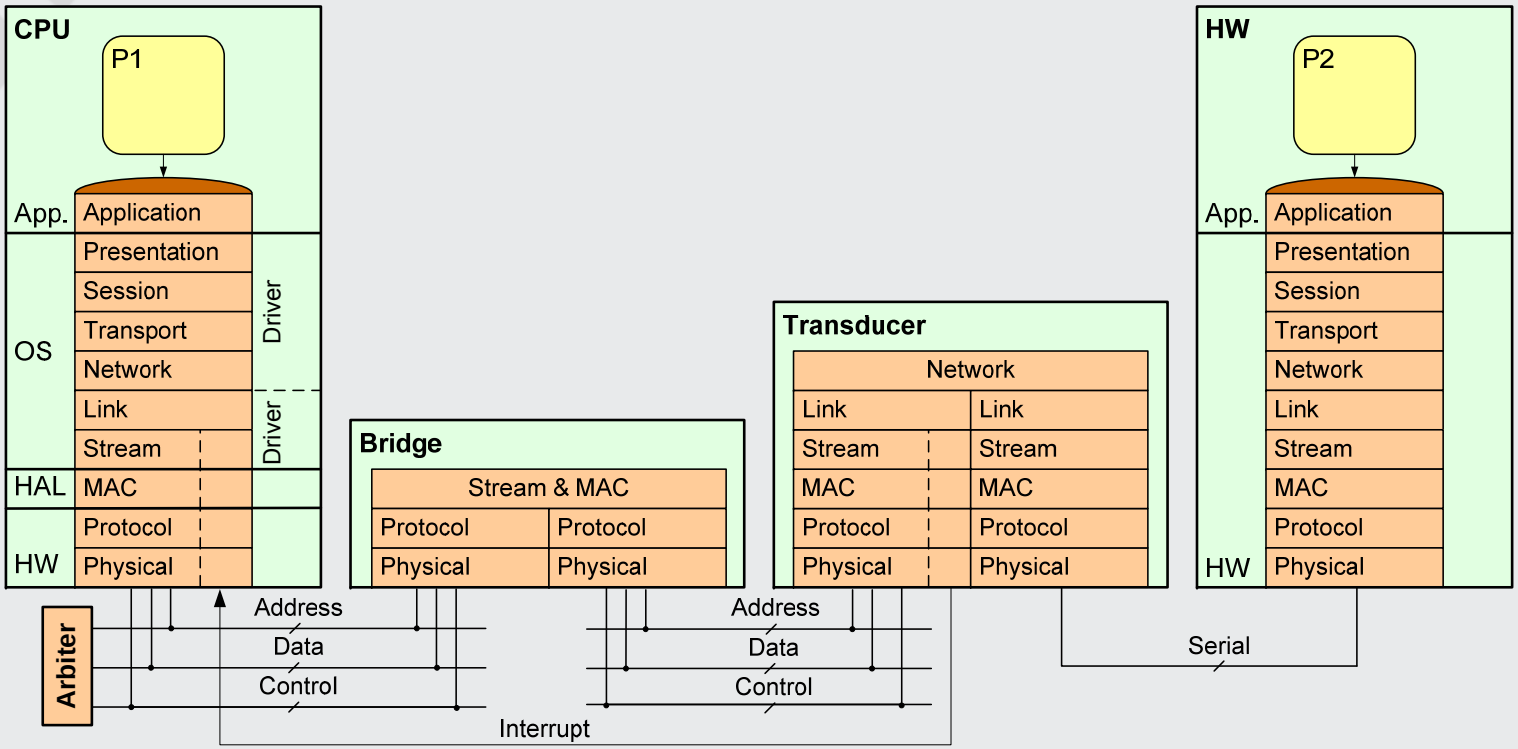
- Topology of communication architecture.
  - ✓ PEs + Memories + CEs
  - ✓ Upper protocol layers inserted into PEs/CEs
  - ✓ Communication via point-to-point links
    - o Synchronous packet transfers (data transfers)
    - o Memory accesses (shared memory, memory-mapped I/O)
    - o Events (control flow)

Protocol TLM



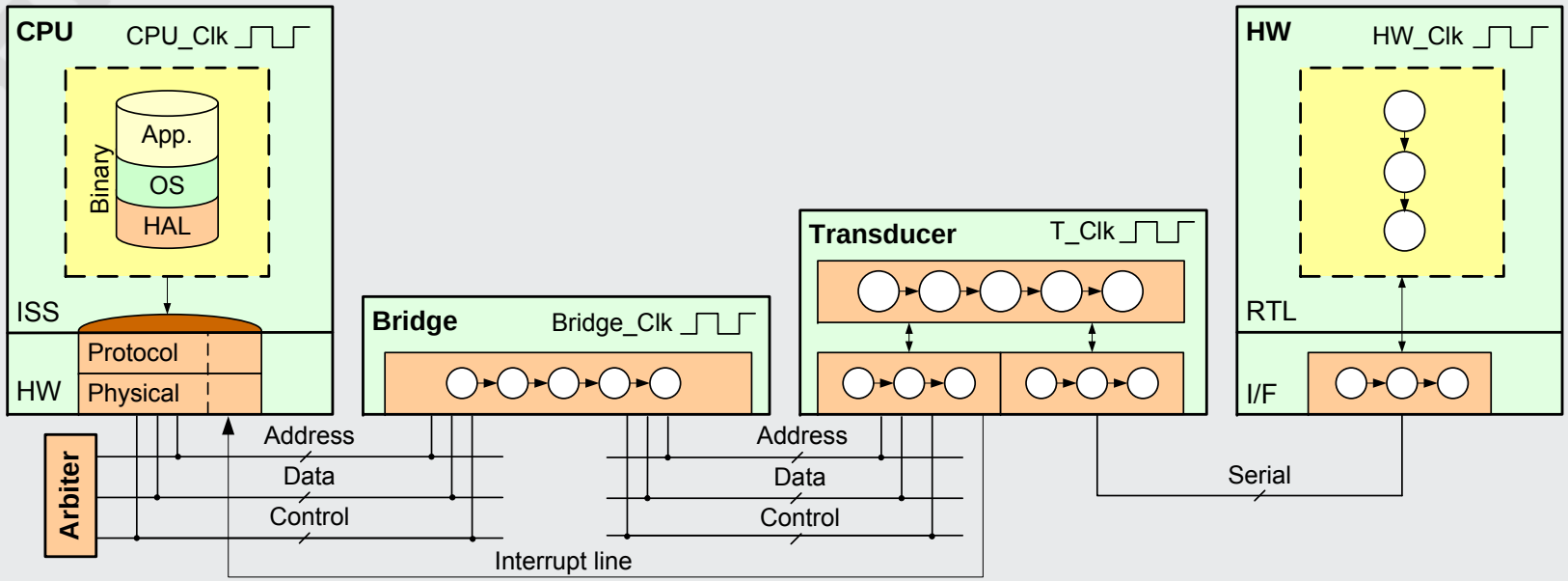
- Abstract component & bus structure/architecture
  - ✓ PEs + Memories + CEs + Busses
  - ✓ Communication layers down to protocol transactions
  - ✓ Communication via transaction-level channels
    - o Bus protocol transactions (data transfers)
    - o Synchronization events (interrupts)

Bus Cycle-Accurate Model (BCAM)



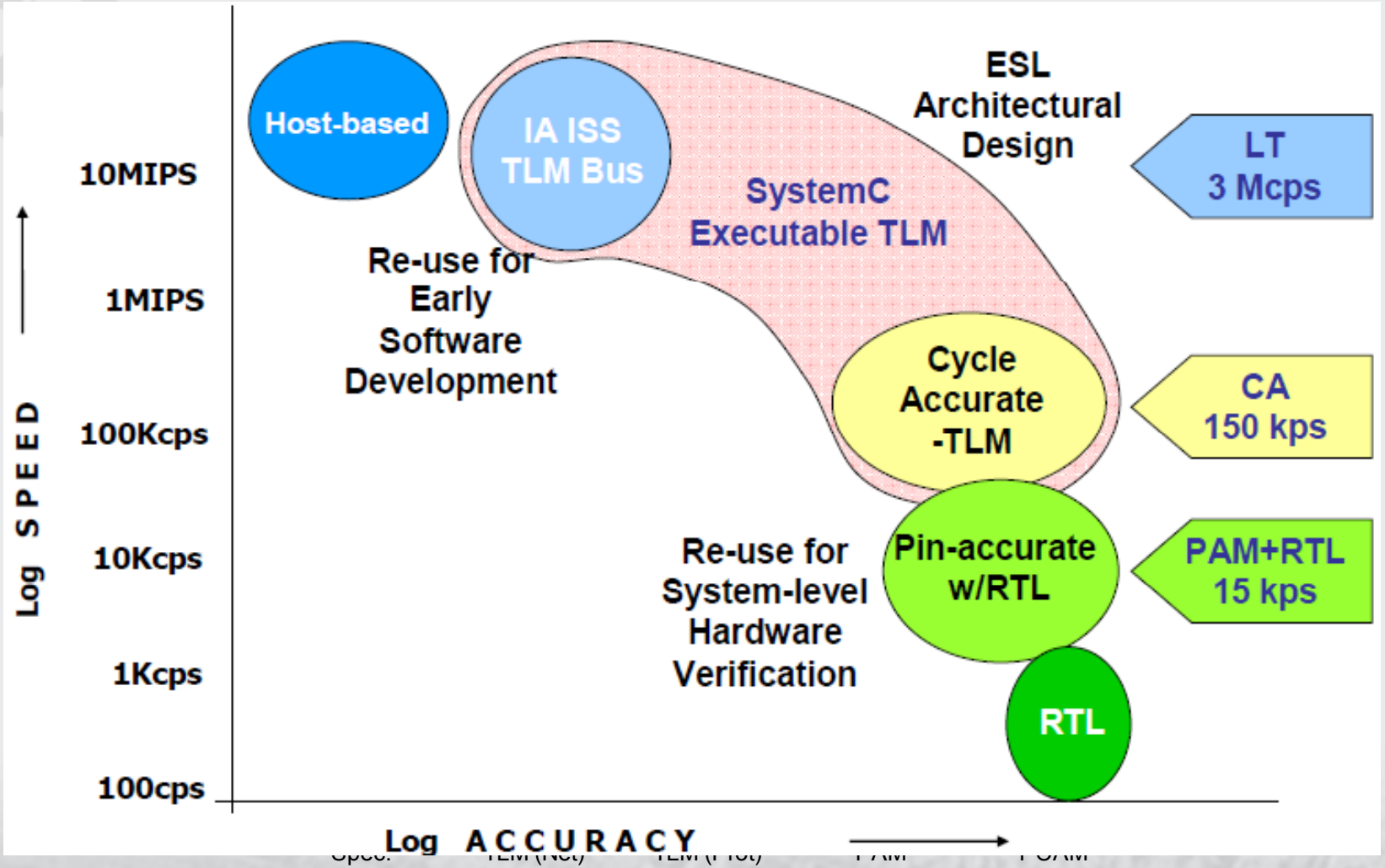
- Component & bus structure/architecture
  - ✓ PEs + Memories + CEs + Busses
  - ✓ Pin-accurate bus-functional components
  - ✓ Pin- and cycle-accurate communication
    - o Bus and interrupt protocols
    - o Pins and wires

Cycle-Accurate Model (CAM)



- Component & bus implemenation
  - ✓ PEs + Memories + CEs + Busses
  - ✓ Cycle-accurate components
    - o Instruction-set simulators (ISS) runing final target binaries
    - o RTL hardware models
    - o Bus protocol state machines

Simulation Speed & Accuracy



Summary and Conclusions

- Modeling of system computation and communication
  - ✓ From specification
    - o System behavior, Models of Computation (MoCs)
  - ✓ To implementation
    - o Layers of implementation detail
  - ✓ Flow of well-defined models as basis for automated design process
- Various level of abstraction, accuracy and speed
  - ✓ Functional specification
    - o Native speeds but inaccurate
  - ✓ Traditional cycle-accurate model (CAM)
    - o 100% accurate but slow
  - ✓ Transaction-level models (TLMs)
    - o Fast and accurate virtual prototyping